Go HTTP Frameworks Deep Dive

A practical, code driven guide to building HTTP servers in Go.

This repository explores **net/http**, **Chi**, **Gin**, **Echo**, **Fiber**, and **Mizu** - by solving the same problems in the same order, using real, runnable programs.

Each topic starts with a minimal working server and then explains what actually happens when a request enters the process and a response leaves it. The focus stays on behavior and execution flow rather than surface level APIs.

The intent is understanding. You should be able to explain why something works, not just how to write it.

Overview

What this repository provides

This repository is a structured walk through how Go HTTP frameworks behave.

It helps you:

- see how different frameworks structure the same application
- follow a request through routing, middleware, handlers, and response writing
- understand ownership of servers, routers, contexts, and lifecycles
- compare tradeoffs based on execution, not marketing

This repository does not aim to benchmark performance or list features.

The emphasis stays on clarity, mechanics, and long term understanding.

Each section stands on its own. You can read from start to finish or jump directly to a topic you care about.

Frameworks covered

Framework	Description	Version	Release date	GitHub
net/http	Go standard library HTTP server	go1.25.5	2025-12-02	https://github.com/golang/go
Chi	Router built on net/http	v5.2.3	2025-08-26	https://github.com/go-chi/chi
Gin	HTTP framework with helpers	v1.11.0	2025-09-20	https://github.com/gin-gonic/gin
Echo	HTTP framework with error returns	v4.14.0	2025-12-11	https://github.com/labstack/echo
Fiber	fasthttp based framework	v2.52.10	2025-11-19	https://github.com/gofiber/fiber
Mizu	net/http based framework	v0.2.2	2025-12-15	https://github.com/go-mizu/mizu

Versions are included for reference. The behaviors discussed here remain stable across minor releases.

How to use this repository

Each topic lives in its own folder.

Inside each folder you will find:

- a README.md that explains the topic step by step
- a main.go file for each framework
- small examples that you can run directly

You do not need to search across folders to follow the explanation. Every README includes the full code it discusses.

A recent Go version is enough to run the examples unless noted otherwise.

Topics and reading order

You can follow the topics in sequence or jump to any section.

Folder	Topic	File
01-hello-world	First HTTP server	01-hello-world/README.md
02-application	Application setup	02-application/README.md
03-handler-signature	Handlers and request flow	03-handler-signature/README.md
04-routing	Paths, methods, and matching	04-routing/README.md
05-route-groups	Grouping routes	05-route-groups/README.md
06-middleware-chain	Middleware order	06-middleware-chain/README.md
07-short-circuit	Early exits	07-short-circuit/README.md
08-error-handling	Errors and panics	08-error-handling/README.md
09-request-input	Reading headers and bodies	09-request-input/README.md
10-response-output	Writing responses	10-response-output/README.md
11-json	JSON handling	11-json/README.md
12-path-params	Path parameters	12-path-params/README.md
13-static-files	Static and embedded files	13-static-files/README.md
14-templates	HTML templates	14-templates/README.md
15-forms-upload	Forms and file uploads	15-forms-upload/README.md
16-websocket	WebSockets	16-websocket/README.md
17-sse	Server-Sent Events	17-sse/README.md
18-context-cancel	Context and cancellation	18-context-cancel/README.md
19-shutdown	Graceful shutdown	19-shutdown/README.md
20-logging	Logging basics	20-logging/README.md
21-metrics-tracing	Metrics and tracing	21-metrics-tracing/README.md
22-testing	Testing handlers	22-testing/README.md
23-performance	Performance considerations	23-performance/README.md
24-interop	Working with net/http	24-interop/README.md
25-decision	Choosing and migrating	25-decision/README.md

How the examples are written

All examples follow the same discipline:

- every example runs as written

- files stay small and focused
- no hidden helpers or magic layers
- the same structure appears across frameworks

This allows you to compare behavior directly without learning a new mental model for each section.

What you will learn

By working through this repository, you will understand:

- how routing decisions are made
- how middleware wraps execution
- how request context flows through a server
- how different error models affect control flow
- how shutdown and cleanup are coordinated
- how close each framework stays to net/http

These details become critical as applications grow, change ownership, or need to integrate with other systems.

Final note

Choosing a framework comes down to responsibility.

Every framework takes some control away from you and gives some structure back. The important part is knowing exactly where that trade is made.

Once you understand those boundaries, switching frameworks becomes a design decision rather than a rewrite.

That understanding is the purpose of this repository.

Hello, world

We implement a minimal HTTP server that listens on port 8080 and responds to `GET /` with plain text `hello, world!`.

Each example shows the full runnable code and then explains what actually happens inside the framework when a request is received. The focus is on ownership, control flow, and how much machinery exists between the socket and your handler.

net/http

`nethttp/main.go`

```
package main
```

```
import (
  "fmt"
  "net/http"
)

func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("GET /", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "hello, world!")
  })

  http.ListenAndServe(":8080", mux)
}
```

When this program starts, `http.ListenAndServe` creates an `http.Server` and binds a TCP listener to port 8080. From this point on, the runtime enters an accept loop where each incoming connection is handled concurrently. The server is responsible for reading raw bytes from the socket, parsing the HTTP request line, headers, and body, and constructing a `*http.Request` value that represents the parsed input.

The `ServeMux` passed to `ListenAndServe` becomes the root `http.Handler`. Once a request has been parsed, the server invokes `ServeHTTP` on the mux. Internally, the mux matches the request method and path against its routing structure and selects the most specific handler. In recent Go versions this is not a naive linear scan, but the important point is that routing happens before any user code runs.

The handler function receives the original `ResponseWriter` and `*http.Request`. There is no intermediate abstraction. Writing to the `ResponseWriter` writes directly to the response stream associated with the connection. The first write implicitly commits the response headers and status code. Once the handler returns, the server finalizes the response and the connection may be reused or closed depending on headers and protocol state.

There is no centralized error handling, no middleware chain, and no shared request context beyond `context.Context` on the request itself. Control flow is explicit and local, and the handler fully owns the response lifecycle.

Chi

chi/main.go

```
package main

import (
  "fmt"
  "net/http"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()
```

```
  r.Get("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "hello, world!")
  })

  http.ListenAndServe(":8080", r)
}
```

Chi inserts itself between net/http and your handler only at the routing layer. The HTTP server still performs all parsing, connection management, and request construction exactly as it does with a plain `ServeMux`. The difference begins when the server calls `ServeHTTP` on the Chi router instead of the standard mux.

Inside Chi, the request path is decomposed into segments and matched against a tree structure built from registered routes. This tree allows Chi to efficiently resolve static paths, parameters, and wildcards. When a match is found, any extracted parameters are stored in the request's context rather than passed as function arguments.

After routing, Chi invokes the handler using the standard `func(http.ResponseWriter, *http.Request)` signature. Response writing is unchanged. The handler still writes directly to the underlying connection through the `ResponseWriter`. Chi does not buffer output or delay writes. It only decides which handler runs and which middleware wraps it.

The result is a router that adds structure and composition without altering the fundamental ownership model of net/http.

Gin

gin/main.go

```
package main

import (
  "net/http"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()

  r.GET("/", func(c *gin.Context) {
    c.String(http.StatusOK, "hello, world!")
  })

  r.Run(":8080")
}
```

Gin changes the execution model by replacing the handler signature and centralizing control in a framework-managed context object. When `r.Run` is called, Gin creates an `http.Server` and installs its engine as the root handler. From net/http's perspective, Gin is just another `http.Handler`.

When a request arrives, Gin's `ServeHTTP` method is invoked. At this point Gin allocates or reuses a `gin.Context` from an internal pool. This context wraps the original `ResponseWriter` and `*http.Request` and also carries routing metadata, middleware state, and response status tracking.

Routing occurs inside Gin's own tree structure, and the resulting handler chain is executed using the shared context. Middleware and handlers all mutate the same context instance. Helper methods such as `c.String` write through Gin's wrapped `ResponseWriter`, allowing Gin to observe status codes and headers as they are set.

Unlike net/http, the handler no longer owns the response directly. Control flow is inverted. Gin owns the request lifecycle, and user code operates inside it by mutating the context.

Echo

`echo/main.go`

```
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
)

func main() {
  e := echo.New()

  e.GET("/", func(c echo.Context) error {
    return c.String(http.StatusOK, "hello, world!")
  })

  e.Start(":8080")
}
```

Echo is similar to Gin in that it introduces a framework context and routing layer, but it differs in one key design choice: handlers return an `error`. This makes error propagation explicit rather than implicit.

When Echo starts, it creates an `http.Server` and registers its internal handler. For each request, Echo creates a context that wraps the `ResponseWriter` and `*http.Request`. The handler is invoked and writes to the response through helper methods on the context.

After the handler returns, Echo inspects the returned error. If it is non-nil, Echo routes execution through a centralized error handler, which decides how to translate the error into an HTTP response. This means error handling is part of the normal control flow rather than an exceptional path.

Response writing still occurs through net/http, but the decision of whether a response represents success or failure is centralized rather than scattered across handlers.

Fiber

`fiber/main.go`

```go
package main

import (
  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()

  app.Get("/", func(c *fiber.Ctx) error {
    return c.SendString("hello, world!")
  })

  app.Listen(":8080")
}
```

Fiber departs entirely from net/http and is built on top of fasthttp. This changes the execution model at a much lower level. Instead of net/http parsing requests and managing connections, fasthttp handles raw socket IO and uses its own request and response types.

When a connection arrives, fasthttp parses the request into its own structures. Fiber then wraps these structures in a `*fiber.Ctx`, which is passed to the handler. The handler mutates the context to set status and body and returns an error.

Contexts are aggressively pooled and reused. This makes allocation cheaper but imposes strict lifetime rules. A context must never be stored or referenced outside the handler, because it will be reused for another request.

Because Fiber does not use net/http types, it does not interoperate directly with net/http middleware or handlers. The ecosystem boundary is hard, not conceptual.

Mizu

`mizu/main.go`

```go
package main

import (
  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()

  app.Get("/", func(c *mizu.Ctx) error {
    return c.Text(200, "hello, world!")
  })

  app.Listen(":8080")
}
```

Mizu stays within the net/http execution model while introducing structured error handling and a consistent context API. When `app.Listen` is called, Mizu creates an `http.Server` and registers itself as the root handler. net/http continues to handle connections, parsing, and request construction.

When a request is received, Mizu's router matches the method and path and allocates or reuses a `*mizu.Ctx`. This context holds the original `ResponseWriter` and `*http.Request`, along with route metadata and middleware state.

Handlers write responses through helper methods on the context, which forward to the underlying `ResponseWriter`. The handler returns an error value. If the error is non-nil, Mizu routes execution through centralized error handling logic that decides how to produce a response.

The key distinction is that Mizu adds structure without leaving the net/http ecosystem. Handlers still operate on real net/http primitives, but control flow is more explicit and uniform.

Direct technical comparison

Framework	Handler signature	Response writing	Error return	Transport
net/http	`func(w, r)`	direct	no	net/http
Chi	`func(w, r)`	direct	no	net/http
Gin	`func(*gin.Context)`	via context	no	net/http
Echo	`func(echo.Context) error`	via context	yes	net/http
Fiber	`func(*fiber.Ctx) error`	via context	yes	fasthttp
Mizu	`func(*mizu.Ctx) error`	via context	yes	net/http

Understanding these internal differences early makes later topics such as middleware order, cancellation, and graceful shutdown much easier to reason about.

Application wiring

This section grows the program slightly without changing what the HTTP endpoint does.

The request handling still responds with `hello, world!`, but the structure stops being accidental. Wiring becomes explicit: a program decides where objects are created, who holds references to them, and which layer owns each responsibility. That decision sets the shape of everything that follows, including configuration, testing, graceful shutdown, observability, and integration with other services.

Wiring answers questions every service eventually faces:

- who creates the HTTP server
- who owns routing
- where routes are registered
- what object handles incoming requests

These differences are about ownership and control. They influence how easily a service can be embedded, how cleanly it can be tested without sockets, and how confidently timeouts and lifecycle can be enforced.

Each subsection links to runnable code and explains how responsibility flows at runtime.

net/http

`nethttp/main.go`

```go
package main

import (
  "fmt"
  "net/http"
)

func main() {
  mux := newRouter()
  srv := &http.Server{
    Addr:    ":8080",
    Handler: mux,
  }

  srv.ListenAndServe()
}

func newRouter() http.Handler {
  mux := http.NewServeMux()

  mux.HandleFunc("GET /", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "hello, world!")
  })

  return mux
}
```

This layout draws a sharp line between process lifecycle and request behavior.

`main` owns lifecycle decisions. It chooses the address, constructs `http.Server`, and sets the server's `Handler`. The server becomes the runtime entry point for every request. That `Handler` field is the only dependency the server needs to do its job: accept connections, parse requests, and dispatch.

`newRouter` owns HTTP behavior. It constructs a router, registers routes, and returns an `http.Handler`. Returning an interface is important. It allows the server layer to depend on a contract rather than a concrete implementation. A different router can be substituted without changing how the server is created.

At runtime, the boundary is a single call: the server invokes `ServeHTTP(w, r)` on the handler it holds. Everything beyond that is a handler graph owned by the application. That makes the control flow easy to reason about.

Key ownership boundaries:

Concern	Owner
sockets, accept loop	`http.Server`
request parsing	`net/http`
routing decision	`ServeMux`
response writing	handler functions

This approach keeps configuration flexible. Timeouts, TLS, listener type, and shutdown orchestration live next to the server because that layer owns them.

Chi

chi/main.go

```
package main

import (
  "fmt"
  "net/http"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := newRouter()
  srv := &http.Server{
    Addr:    ":8080",
    Handler: r,
  }

  srv.ListenAndServe()
}

func newRouter() http.Handler {
  r := chi.NewRouter()

  r.Get("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "hello, world!")
  })

  return r
}
```

The wiring remains the same shape as net/http: `main` creates the server, the router is injected via `Handler`, and the server calls `ServeHTTP`.

The significant detail is that the Chi router satisfies `http.Handler`, so the server stays unaware of Chi. That preserves a stable integration surface. Anything that expects an `http.Handler` can wrap or host this router: reverse proxies, instrumenting middleware, test servers, and other routers.

At runtime, the call graph still begins in net/http and crosses into the application at `ServeHTTP`. Chi performs route matching inside that call and then invokes the registered handler.

Ownership stays clean:

Concern	Owner
server lifecycle	`main` via `http.Server`
routing implementation	Chi router
handler API	net/http handler signature

This makes it easy to swap routing implementations without changing how the service binds ports or configures timeouts.

Gin

`gin/main.go`

```
package main

import (
  "net/http"

  "github.com/gin-gonic/gin"
)

func main() {
  r := newRouter()
  r.Run(":8080")
}

func newRouter() *gin.Engine {
  r := gin.New()

  r.GET("/", func(c *gin.Context) {
    c.String(http.StatusOK, "hello, world!")
  })

  return r
}
```

Gin shifts where lifecycle responsibilities live. The application creates a `*gin.Engine`, and `Run` takes over server startup. That changes the ownership boundary: the framework owns the act of constructing and running the server, and user code interacts with that boundary through Gin's APIs.

This wiring trades explicit server construction for a single call. The router and the server lifecycle are tightly coupled by default.

Two consequences follow from that coupling:

- server configuration is driven through framework knobs or alternative startup paths, rather than by directly constructing `http.Server` in `main`
- the entry point for requests moves behind `Run`, which installs the Gin engine as the server handler and starts listening

At runtime, net/http still performs connection handling and parsing, then calls the Gin engine's `ServeHTTP`. The difference is where that server is created and configured.

Ownership boundary shifts:

Concern	Owner
server creation	Gin via `Run`
server configuration	Gin-first unless bypassed
routing + middleware	Gin engine
request pipeline	Gin context chain

This model reduces boilerplate and standardizes structure, while moving more control into framework conventions.

Echo

`echo/main.go`

```
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
)

func main() {
  e := newApp()
  e.Start(":8080")
}

func newApp() *echo.Echo {
  e := echo.New()

  e.GET("/", func(c echo.Context) error {
    return c.String(http.StatusOK, "hello, world!")
  })
```

```
  return e
}
```

Echo follows a similar shape to Gin: an application object owns routing and middleware, and a method on that object starts the server. The wiring presents the framework object as the primary unit of composition.

The server boundary is created by `Start`. The user does not construct `http.Server` directly in this shape. That pushes lifecycle configuration closer to framework configuration, with the application object becoming the place where request behavior and cross-cutting concerns accumulate.

At runtime, Echo installs its handler into the server and dispatches through its internal router and middleware layers. The handler signature returns `error`, which becomes important later because failures can bubble into centralized handling rather than being written directly in each handler.

Ownership boundary:

Concern	Owner
server startup	Echo via `Start`
routing	Echo router
error path	central dispatcher + error handler

This tends to feel cohesive as applications grow because the framework object becomes the single place where policy and behavior are assembled.

Fiber

fiber/main.go

```
package main

import (
  "github.com/gofiber/fiber/v2"
)

func main() {
  app := newApp()
  app.Listen(":8080")
}

func newApp() *fiber.App {
  app := fiber.New()

  app.Get("/", func(c *fiber.Ctx) error {
    return c.SendString("hello, world!")
  })

  return app
}
```

Fiber owns the runtime end-to-end. The service does not present an `http.Handler` boundary and does not use `http.Server` because Fiber uses a fasthttp-based stack.

Wiring is compact because the framework object covers routing, request parsing, and the listen loop. Calling `Listen` enters Fiber's server loop.

That changes the integration surface:

- middleware and tooling designed for net/http do not plug in directly
- embedding Fiber inside a larger net/http server requires adapters
- the request and response types come from Fiber's ecosystem

At runtime, request parsing and dispatch happen entirely inside Fiber's engine. The program hands control to Fiber, and Fiber invokes handlers via its own context type.

Ownership boundary:

Concern	Owner
accept loop + parsing	Fiber/fasthttp
routing	Fiber
handler API	Fiber context

This model tends to prioritize a cohesive internal pipeline over interoperability.

Mizu

`mizu/main.go`

```go
package main

import (
  "github.com/go-mizu/mizu"
)

func main() {
  app := newApp()
  app.Listen(":8080")
}

func newApp() *mizu.App {
  app := mizu.New()

  app.Get("/", func(c *mizu.Ctx) error {
    return c.Text(200, "hello, world!")
  })

  return app
}
```

Mizu presents an application object as the place where routing and middleware are assembled, then exposes a `Listen` method to start serving. Internally, the serving model aligns with net/http, which preserves the standard library's execution model while offering a framework-style surface.

The wiring separates concerns in practice:

- the app object gathers routes and middleware
- listening creates and runs the server using net/http semantics

That means the entry point remains compatible with net/http style request execution, and the service can interact with net/http concepts like handlers, servers, and middleware wrappers in a predictable way.

Ownership boundary:

Concern	Owner
routing + middleware registration	Mizu app object
connection handling	net/http server underneath
error flow	handler error return into centralized handling

This tends to keep integration options open while still giving a structured application object to build on.

What to learn from this section

Wiring choices determine where control lives.

- net/http and Chi keep a visible `http.Server` boundary in user code, which makes lifecycle configuration straightforward and explicit
- Gin and Echo make the framework object the center of gravity, often starting the server through a framework call
- Fiber takes full ownership by leaving the net/http ecosystem
- Mizu uses an app object while keeping a net/http-aligned execution model underneath

These choices affect:

- configuring timeouts, TLS, and custom listeners
- testing routing and middleware without binding ports
- embedding one service inside another
- attaching ecosystem middleware and instrumentation

Handler signatures and request lifetime

This section explains what a handler actually represents in each framework and how a single HTTP request moves through your program from arrival to completion.

At first glance, handlers across frameworks appear similar. They all respond to requests and write responses. The important differences appear when you look at *where data lives*, *who owns the response*, *how errors are expressed*, and *how long objects remain valid*. These details shape how middleware works, how code composes, how testing feels, and how easy it is to reason about failures.

A handler signature is a contract. It defines which objects are created by the server, which are owned by the framework, which are safe to keep, and which must never escape the request. Once this contract is understood, most higher-level framework behavior becomes predictable.

Each subsection below shows the full runnable program and then walks through the request lifetime using concrete mechanics, small snippets, and focused comparisons.

net/http

`nethttp/main.go`

```
package main

import (
  "fmt"
  "net/http"
)

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("GET /", handler)
  http.ListenAndServe(":8080", mux)
}

func handler(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "hello, world!")
}
```

In net/http, the handler is a direct function call from the server into user code. After the TCP connection is accepted and the request is parsed, the server allocates two objects and passes them to the handler: a `*http.Request` and a `ResponseWriter`. No additional abstraction is involved.

The request object represents parsed input. Its fields are populated before the handler runs. The body is a stream tied to the underlying connection. Reading from `r.Body` affects connection reuse. Leaving unread data can prevent keep-alive. Draining or closing the body early changes server behavior.

The response writer represents a stateful output stream. It tracks whether headers have been committed and writes bytes back to the client. The first call to `Write` implicitly commits headers if `WriteHeader` was not called explicitly.

```
w.WriteHeader(200)
w.Write([]byte("hello"))
```

or simply:

```
fmt.Fprintln(w, "hello")
```

There is no return value. The handler communicates intent only through side effects. Errors are expressed by writing an error response, returning without writing, or panicking.

The lifetime model is strict and simple. The server calls the handler and expects all request-scoped work to finish before the function returns. Any goroutine that outlives the handler must copy everything it needs.

Key properties:

Aspect	net/http
Handler type	func(w, r)
Response ownership	direct
Error signaling	side effects, panic
Object reuse	none

Chi

chi/main.go

```
package main

import (
  "fmt"
  "net/http"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()
  r.Get("/", handler)
  http.ListenAndServe(":8080", r)
}

func handler(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "hello, world!")
}
```

Chi keeps the same handler contract as net/http and moves framework logic into routing and middleware composition. The server still calls `ServeHTTP` with a `ResponseWriter` and a `*http.Request`.

Before invoking the handler, Chi matches the request path and method against its routing tree. Route parameters and metadata are attached to the request context.

```
id := chi.URLParam(r, "id")
```

From the handler's perspective, nothing else changes. Response writing follows the same commit rules. Errors propagate in the same way.

The lifetime of objects remains identical to net/http. Chi does not pool or reuse handler-level objects. Middleware wraps handlers using standard function composition.

Key properties:

Aspect	Chi
Handler type	func(w, r)
Response ownership	direct
Error signaling	side effects, panic
Request extension	via context
Object reuse	none

Gin

`gin/main.go`

```
package main

import (
  "net/http"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()
  r.GET("/", handler)
  r.Run(":8080")
}

func handler(c *gin.Context) {
  c.String(http.StatusOK, "hello, world!")
}
```

Gin replaces the handler contract with a framework-managed context. When a request arrives, Gin allocates or reuses a `gin.Context` from a pool and binds it to the current request and response writer.

The context contains both input and output state:

```
c.Request        // *http.Request
c.Writer         // wrapped ResponseWriter
c.Params         // route params
```

Handlers do not write to the response directly. Instead, they call helper methods that mutate the context and forward writes through a wrapped writer.

```
c.JSON(200, obj)
c.String(200, "ok")
```

Control flow is managed by Gin. Middleware and handlers execute in a linear chain controlled by an internal index. Early exits occur by mutating the context rather than by returning values.

Context pooling defines the lifetime rule. After the request completes, the context is reset and returned to the pool. Any reference to the context after the handler returns is unsafe.

Key properties:

Aspect	Gin
Handler type	func(*Context)
Response ownership	via context
Error signaling	context state, panic
Object reuse	pooled context

Echo

echo/main.go

```
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
)

func main() {
  e := echo.New()
  e.GET("/", handler)
  e.Start(":8080")
}

func handler(c echo.Context) error {
  return c.String(http.StatusOK, "hello, world!")
}
```

Echo introduces a return value to express control flow. The handler returns an error that represents success or failure.

The context wraps request and response objects and exposes helper methods. Writing output still happens through helpers, but the outcome is explicit.

```
return c.JSON(200, data)
```

or:

```
return echo.NewHTTPError(400, "bad request")
```

After the handler returns, Echo inspects the error. A non-nil error triggers centralized error handling, which decides how to write the response.

The context is request-scoped and not reused in unsafe ways. Execution paths are easier to follow because success and failure are expressed as values rather than inferred.

Key properties:

Aspect	Echo
Handler type	func(Context) error
Response ownership	via context
Error signaling	return value
Object reuse	request-scoped

Fiber

fiber/main.go

```
package main

import (
  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()
  app.Get("/", handler)
  app.Listen(":8080")
}

func handler(c *fiber.Ctx) error {
  return c.SendString("hello, world!")
}
```

Fiber uses a similar signature to Echo but operates on top of fasthttp rather than net/http. Requests and responses are represented by fasthttp structures, and the context holds pointers into those structures.

When a request arrives, Fiber retrieves a context from a pool, routes the request, invokes the handler, then resets the context.

```
body := c.Body()   // byte slice reused later
```

Buffers and slices returned from the context are reused aggressively. Keeping references beyond the handler can lead to corrupted data.

Error returns feed Fiber's error handling pipeline. Response writing occurs by mutating the fasthttp response and letting the server flush it after the handler returns.

Key properties:

Aspect	Fiber
Handler type	func(*Ctx) error
Response ownership	via context
Error signaling	return value
Object reuse	pooled context and buffers

Mizu

mizu/main.go

```
package main

import (
  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()
  app.Get("/", handler)
  app.Listen(":8080")
}

func handler(c *mizu.Ctx) error {
  return c.Text(200, "hello, world!")
}
```

Mizu combines explicit error returns with net/http compatibility. The context wraps `http.ResponseWriter` and `*http.Request` and carries routing and middleware state.

Handlers write responses through helper methods and return an error to signal failure. Centralized error handling converts errors into responses consistently.

```
return c.JSON(200, data)
```

or:

```
return mizu.ErrBadRequest
```

The request lifetime stays aligned with net/http. Parsing, cancellation, and response commit semantics come from the standard library. The context is request-scoped and designed to remain safe as long as handlers respect request boundaries.

Key properties:

Aspect	Mizu
Handler type	func(*Ctx) error
Response ownership	via context
Error signaling	return value
Object reuse	controlled

Summary

Framework	Handler type	Error channel	Response ownership	Reuse model
net/http	func(w, r)	side effects, panic	direct	none
Chi	func(w, r)	side effects, panic	direct	none
Gin	func(*Context)	context state	indirect	pooled
Echo	func(Context) error	return value	indirect	request-scoped
Fiber	func(*Ctx) error	return value	indirect	pooled
Mizu	func(*Ctx) error	return value	indirect	controlled

Once the handler contract and lifetime rules are clear, the behavior of middleware, testing patterns, cancellation, and shutdown becomes much easier to reason about across frameworks.

Routing: paths, methods, and precedence

Routing determines how an incoming HTTP request is translated into a specific piece of code. By the time routing runs, the server has already accepted a connection, parsed the request line and headers, and constructed the request object. Routing sits between request parsing and handler execution and decides what code will run next, if any.

This decision sounds simple, but it carries several consequences. The routing model defines how paths are compared, where HTTP methods are checked, how conflicts are resolved when multiple routes could apply, and whether redirects or rewrites happen automatically. It also determines how much work happens before your handler is reached and how predictable the matching rules are as a codebase grows.

At this stage, handler signatures stay fixed. The focus shifts entirely to how frameworks organize routing logic and how that logic behaves under overlap, ambiguity, and scale.

Each section below links to the runnable code, shows the full `main.go`, and then explains how routing behaves internally with concrete mechanics and examples.

net/http

[nethttp/main.go]

```go
package main

import (
  "fmt"
  "net/http"
)

func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("GET /", root)
  mux.HandleFunc("GET /users", users)
  mux.HandleFunc("GET /users/", userSubtree)

  http.ListenAndServe(":8080", mux)
}

func root(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "root")
}

func users(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "users")
}

func userSubtree(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "users subtree:", r.URL.Path)
}
```

Routing in `http.ServeMux` is based on pattern rules rather than an explicit routing tree. Each registered pattern is a string that participates in precedence comparison. When a request arrives, the mux selects the best match according to a small set of ordering rules that determine which handler wins.

Two properties dominate precedence. Exact matches outrank subtree matches, and longer patterns outrank shorter ones. A pattern ending with a slash represents a subtree and matches any path that shares the prefix.

In the example above, three patterns coexist:

```
GET /users        exact
GET /users/       subtree
GET /             subtree
```

A request to `/users` selects the exact match. A request to `/users/42` selects the subtree `/users/`. A request to `/anything-else` falls back to `/`.

Method matching is part of the pattern string. The mux checks the HTTP method before calling the handler. Redirect behavior for trailing slashes is handled internally by the mux and occurs before handler execution.

There is no parameter extraction step. The handler receives the raw path and must parse it manually if needed. Routing remains simple and predictable, but expressive power stays limited.

Key routing characteristics:

Aspect	net/http
Matching model	pattern rules
Method check	built into mux
Precedence	exact, then longest
Parameters	none
Redirects	implicit

Chi

chi/main.go

```
package main

import (
  "fmt"
  "net/http"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()

  r.Get("/", root)
  r.Get("/users", users)
  r.Get("/users/{id}", userByID)

  http.ListenAndServe(":8080", r)
}

func root(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "root")
}

func users(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "users")
```

```
}

func userByID(w http.ResponseWriter, r *http.Request) {
  id := chi.URLParam(r, "id")
  fmt.Fprintln(w, "user:", id)
}
```

Chi organizes routes into a radix-style tree per HTTP method. Each path is split into segments, and each segment becomes a node. Static segments and parameter segments occupy different positions in the tree, which allows Chi to enforce clear precedence rules.

When a request arrives, Chi selects the tree for the request method and walks it segment by segment. Static segments are matched first, followed by parameter segments. When a parameter segment matches, the value is captured and stored in the request context.

```
id := chi.URLParam(r, "id")
```

Route resolution completes before the handler is called. The router determines the handler and enriches the request context with routing metadata. Handler execution then proceeds using the standard net/http contract.

Redirects are not automatic. Trailing slash behavior must be configured explicitly, which keeps routing behavior predictable and avoids implicit rewrites.

Key routing characteristics:

Aspect	Chi
Matching model	radix tree
Method check	router
Precedence	static > param
Parameters	request context
Redirects	explicit

Gin

gin/main.go

```
package main

import (
  "net/http"

  "github.com/gin-gonic/gin"
)

func main() {
```

```
  r := gin.New()

  r.GET("/", root)
  r.GET("/users", users)
  r.GET("/users/:id", userByID)

  r.Run(":8080")
}

func root(c *gin.Context) {
  c.String(http.StatusOK, "root")
}

func users(c *gin.Context) {
  c.String(http.StatusOK, "users")
}

func userByID(c *gin.Context) {
  id := c.Param("id")
  c.String(http.StatusOK, "user: %s", id)
}
```

Gin uses a tree structure derived from `httprouter`, with separate trees per HTTP method. Routing begins by selecting the tree for the request method, which eliminates method mismatches early.

Path matching proceeds segment by segment. Static segments are matched before parameter segments. When a parameter node matches, the value is written directly into the request context.

Routing and execution are tightly coupled. The router does more than select a handler. It prepares the execution context, binds parameters, and sets up the middleware chain that will run around the handler.

Trailing slash redirects are enabled by default. Requests to `/users/` may be redirected to `/users`, depending on configuration.

Key routing characteristics:

Aspect	Gin
Matching model	tree
Method check	router
Precedence	static > param
Parameters	context
Redirects	default

Echo

`echo/main.go`

```go
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
)

func main() {
  e := echo.New()

  e.GET("/", root)
  e.GET("/users", users)
  e.GET("/users/:id", userByID)

  e.Start(":8080")
}

func root(c echo.Context) error {
  return c.String(http.StatusOK, "root")
}

func users(c echo.Context) error {
  return c.String(http.StatusOK, "users")
}

func userByID(c echo.Context) error {
  return c.String(http.StatusOK, "user: "+c.Param("id"))
}
```

Echo builds and maintains its own routing tree. The router resolves a request to a handler along with the middleware chain that should execute for that route.

Routing determines which code should run next, but it does not perform response writing itself. Execution proceeds through a central dispatcher that runs middleware, invokes the handler, and processes the returned error.

Parameters are stored on the context and retrieved by name. Redirect behavior is configurable and typically disabled unless enabled explicitly.

Key routing characteristics:

Aspect	Echo
Matching model	tree
Method check	router
Precedence	static > param
Parameters	context
Redirects	configurable

Fiber

fiber/main.go

```
package main

import (
  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()

  app.Get("/", root)
  app.Get("/users", users)
  app.Get("/users/:id", userByID)

  app.Listen(":8080")
}

func root(c *fiber.Ctx) error {
  return c.SendString("root")
}

func users(c *fiber.Ctx) error {
  return c.SendString("users")
}

func userByID(c *fiber.Ctx) error {
  return c.SendString("user: " + c.Params("id"))
}
```

Fiber delegates routing to a fasthttp-based router optimized for low allocation and fast matching. Method and path matching happen together at the routing layer.

When a route matches, parameter values are written directly into the request context. Because contexts and buffers are reused, parameter values must be consumed during handler execution or copied out.

Routing completes before handler execution, and the handler runs immediately after matching.

Key routing characteristics:

Aspect	Fiber
Matching model	tree
Method check	router
Precedence	static > param
Parameters	context
Redirects	configurable

Mizu

mizu/main.go

```
package main

import (
  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()

  app.Get("/", root)
  app.Get("/users", users)
  app.Get("/users/:id", userByID)

  app.Listen(":8080")
}

func root(c *mizu.Ctx) error {
  return c.Text(200, "root")
}

func users(c *mizu.Ctx) error {
  return c.Text(200, "users")
}

func userByID(c *mizu.Ctx) error {
  return c.Text(200, "user: "+c.Param("id"))
}
```

Mizu implements routing on top of net/http while using a tree-based matcher. Method and path matching occur before handler execution. Parameters are captured explicitly and stored on the request-scoped context.

Routing resolves a handler and prepares middleware execution. The routing layer decides what should run, while execution remains a separate concern handled by the dispatcher and error pipeline.

Redirect behavior stays explicit. No automatic rewrites occur unless configured.

Key routing characteristics:

Aspect	Mizu
Matching model	tree
Method check	router
Precedence	static > param
Parameters	context
Redirects	explicit

Routing differences that matter

Framework	Matching model	Method handling	Parameters	Redirect behavior
net/http	pattern rules	mux	none	implicit
Chi	radix tree	router	context	explicit
Gin	tree	router	context	default
Echo	tree	router	context	configurable
Fiber	tree	router	context	configurable
Mizu	tree	router	context	explicit

Why this matters

Routing choices affect performance under load, correctness with overlapping paths, and how safely large route sets can evolve. Understanding precedence rules and matching behavior prevents subtle bugs and makes refactoring predictable as applications grow.

Route groups and composition

Route groups help avoid repetition once an API grows beyond a few endpoints. They solve two practical problems: attaching a shared path prefix like `/api/v1`, and attaching shared behavior like authentication for `/admin`. The interesting part sits under the surface. Some stacks implement groups as runtime composition, others treat them as route-registration helpers that precompute final paths and handler chains, and net/http pushes you toward building the mechanism yourself by composing handlers.

This section stays focused on two things only:

- grouping by path prefix, like `/api/v1`

- composing shared behavior, like authentication for `/admin`

net/http

nethttp/main.go

```
package main

import (
  "fmt"
  "net/http"
)

func main() {
  root := http.NewServeMux()

  root.HandleFunc("GET /", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "root")
  })

  apiV1 := http.NewServeMux()
  apiV1.HandleFunc("GET /users", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "api v1 users")
  })
  apiV1.HandleFunc("GET /users/{id}", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "api v1 user:", r.PathValue("id"))
  })

  admin := http.NewServeMux()
  admin.HandleFunc("GET /dashboard", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "admin dashboard")
  })

  root.Handle("/api/v1/", http.StripPrefix("/api/v1", apiV1))
  root.Handle("/admin/", chain(http.StripPrefix("/admin", admin),
requireToken("letmein")))

  http.ListenAndServe(":8080", root)
}

type Middleware func(http.Handler) http.Handler

func chain(h http.Handler, m ...Middleware) http.Handler {
  for i := len(m) - 1; i >= 0; i-- {
    h = m[i](h)
  }
  return h
}

func requireToken(token string) Middleware {
  return func(next http.Handler) http.Handler {
```

```
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
      if r.Header.Get("X-Admin-Token") != token {
        w.WriteHeader(http.StatusUnauthorized)
        w.Write([]byte("unauthorized"))
        return
      }
      next.ServeHTTP(w, r)
    })
  }
}
```

net/http provides routing and handler composition, but no explicit group abstraction. Grouping emerges by mounting sub-routers under a prefix. The root mux performs the outer match, then delegates to a sub-mux. `http.StripPrefix` performs a rewrite so that the mounted mux can match routes as if it lived at `/`.

The effective behavior can be described as a two-step match:

Client path	Root match	Path seen by sub-mux
`/api/v1/users`	`/api/v1/`	`/users`
`/api/v1/users/42`	`/api/v1/`	`/users/42`
`/admin/dashboard`	`/admin/`	`/dashboard`

Middleware composition happens by wrapping handlers. The admin group shows the standard net/http pattern: the group is the handler tree under `/admin/`, and shared behavior is wrapped around that handler tree. The middleware decides whether to call `next.ServeHTTP`.

```
root.Handle("/admin/",
  chain(
    http.StripPrefix("/admin", admin),
    requireToken("letmein"),
  ),
)
```

The main sharp edge is that the sub-mux sees a rewritten path. Any code that logs `r.URL.Path`, constructs redirects, or performs path-based authorization inside the sub-mux operates on the stripped path. If you need the original path, you must preserve it explicitly.

Chi

chi/main.go

```
package main

import (
  "fmt"
  "net/http"
```

```
  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()

  r.Get("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "root")
  })

  r.Route("/api/v1", func(r chi.Router) {
    r.Get("/users", func(w http.ResponseWriter, r *http.Request) {
      fmt.Fprintln(w, "api v1 users")
    })
    r.Get("/users/{id}", func(w http.ResponseWriter, r *http.Request) {
      fmt.Fprintln(w, "api v1 user:", chi.URLParam(r, "id"))
    })
  })

  r.Route("/admin", func(r chi.Router) {
    r.Use(requireToken("letmein"))
    r.Get("/dashboard", func(w http.ResponseWriter, r *http.Request) {
      fmt.Fprintln(w, "admin dashboard")
    })
  })

  http.ListenAndServe(":8080", r)
}

func requireToken(token string) func(http.Handler) http.Handler {
  return func(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
      if r.Header.Get("X-Admin-Token") != token {
        w.WriteHeader(http.StatusUnauthorized)
        w.Write([]byte("unauthorized"))
        return
      }
      next.ServeHTTP(w, r)
    })
  }
}
```

Chi's `Route` provides a scoped router view. It applies a prefix and allows middleware to be attached within that scope. Unlike the net/http approach, no path rewriting is required. The incoming request path remains unchanged, and the router maintains internal state about the current prefix while walking the routing tree.

That distinction matters when building redirects and logs. Inside handlers, `r.URL.Path` remains the client path. Parameter extraction also stays consistent because Chi attaches route state to the request context rather than rewriting the URL.

Middleware inheritance composes predictably. Middleware registered outside the group wraps everything inside. Middleware registered inside the group wraps only the group routes. The resulting execution order follows the router scope structure rather than registration tricks.

A compact mental model:

Concept	Chi group behavior
Prefix	applied logically in the router
Middleware	stacked by scope
Request path	preserved
Composition	runtime wrapper chain

Gin

gin/main.go

```
package main

import (
  "net/http"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()

  r.GET("/", func(c *gin.Context) {
    c.String(http.StatusOK, "root")
  })

  api := r.Group("/api/v1")
  {
    api.GET("/users", func(c *gin.Context) {
      c.String(http.StatusOK, "api v1 users")
    })
    api.GET("/users/:id", func(c *gin.Context) {
      c.String(http.StatusOK, "api v1 user: %s", c.Param("id"))
    })
  }

  admin := r.Group("/admin", requireToken("letmein"))
  {
    admin.GET("/dashboard", func(c *gin.Context) {
      c.String(http.StatusOK, "admin dashboard")
    })
  }
```

```
  r.Run(":8080")
}

func requireToken(token string) gin.HandlerFunc {
  return func(c *gin.Context) {
    if c.GetHeader("X-Admin-Token") != token {
      c.AbortWithStatusJSON(http.StatusUnauthorized, gin.H{"error": "unauthorized"})
      return
    }
    c.Next()
  }
}
```

Gin groups are `RouterGroup` objects carrying a base path and a middleware list. Route registration through a group combines these pieces into a final route entry. The important technical detail is that the prefix and middleware chain are mostly resolved at registration time rather than at request time.

You can think of Gin's group as a route builder:

- prefix is concatenated into the final path matcher
- middleware slices are concatenated into the final handler chain

A useful way to visualize what gets stored:

Registration call	Effective route	Effective handlers
`api.GET("/users", H)`	`/api/v1/users`	`global... + api... + H`
`admin.GET("/dashboard", H)`	`/admin/dashboard`	`global... + admin... + H`

At runtime, dispatch selects a route and runs a precomputed handler chain. Control flow is driven by the framework context. Middleware uses `c.Next()` to continue and `Abort...` to stop. That changes how you reason about early-exit compared to net/http wrappers, since the "call next" decision happens through framework control flow instead of direct function calls.

Echo

[echo/main.go]

```
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
)

func main() {
  e := echo.New()
```

```
  e.GET("/", func(c echo.Context) error {
    return c.String(http.StatusOK, "root")
  })

  api := e.Group("/api/v1")
  api.GET("/users", func(c echo.Context) error {
    return c.String(http.StatusOK, "api v1 users")
  })
  api.GET("/users/:id", func(c echo.Context) error {
    return c.String(http.StatusOK, "api v1 user: "+c.Param("id"))
  })

  admin := e.Group("/admin", requireToken("letmein"))
  admin.GET("/dashboard", func(c echo.Context) error {
    return c.String(http.StatusOK, "admin dashboard")
  })

  e.Start(":8080")
}

func requireToken(token string) echo.MiddlewareFunc {
  return func(next echo.HandlerFunc) echo.HandlerFunc {
    return func(c echo.Context) error {
      if c.Request().Header.Get("X-Admin-Token") != token {
        return echo.NewHTTPError(http.StatusUnauthorized, "unauthorized")
      }
      return next(c)
    }
  }
}
```

Echo groups combine a prefix with a middleware slice and produce a scoped router. The request-time dispatcher resolves the route and then executes middleware in hierarchical order. Middleware composition uses function wrapping where both middleware and handler return an error. That makes early exit precise: returning an error stops the chain and hands control to the centralized error handler.

This leads to a clear group behavior:

- prefix selects a route subset
- middleware is scoped to that subset
- failure propagates as an error value

A small contrast with Gin helps here:

Stack	“Stop execution” looks like
Gin	`c.Abort...` inside context-driven chain
Echo	`return err` inside wrapper chain

Fiber

fiber/main.go

```
package main

import (
  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()

  app.Get("/", func(c *fiber.Ctx) error {
    return c.SendString("root")
  })

  api := app.Group("/api/v1")
  api.Get("/users", func(c *fiber.Ctx) error {
    return c.SendString("api v1 users")
  })
  api.Get("/users/:id", func(c *fiber.Ctx) error {
    return c.SendString("api v1 user: " + c.Params("id"))
  })

  admin := app.Group("/admin", requireToken("letmein"))
  admin.Get("/dashboard", func(c *fiber.Ctx) error {
    return c.SendString("admin dashboard")
  })

  app.Listen(":8080")
}

func requireToken(token string) fiber.Handler {
  return func(c *fiber.Ctx) error {
    if c.Get("X-Admin-Token") != token {
      return c.Status(401).SendString("unauthorized")
    }
    return c.Next()
  }
}
```

Fiber groups work as prefix-based registration helpers and middleware scoping tools. Registration through a group combines prefix and route path into a final matcher, and group middleware is attached to the routes registered under that group.

Middleware flow uses `c.Next()` to continue. The underlying context is pooled and fasthttp-backed, so group middleware must treat all request-derived values as request-bound. Anything stored beyond handler execution must be copied.

A simplified view of what gets produced:

Registration	Effective route	Group behavior
`api.Get("/users", ...)`	`/api/v1/users`	prefix concatenation
`admin := app.Group("/admin", mw)`	`/admin/...`	group-scoped middleware

Mizu

mizu/main.go

```
package main

import (
  "net/http"

  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()

  app.Get("/", func(c *mizu.Ctx) error {
    return c.Text(http.StatusOK, "root")
  })

  api := app.Group("/api/v1")
  api.Get("/users", func(c *mizu.Ctx) error {
    return c.Text(http.StatusOK, "api v1 users")
  })
  api.Get("/users/:id", func(c *mizu.Ctx) error {
    return c.Text(http.StatusOK, "api v1 user: "+c.Param("id"))
  })

  admin := app.Group("/admin")
  admin.Use(requireToken("letmein"))
  admin.Get("/dashboard", func(c *mizu.Ctx) error {
    return c.Text(http.StatusOK, "admin dashboard")
  })

  app.Listen(":8080")
}

func requireToken(token string) mizu.Middleware {
  return func(next mizu.Handler) mizu.Handler {
    return func(c *mizu.Ctx) error {
      if c.Request().Header.Get("X-Admin-Token") != token {
        return c.Text(http.StatusUnauthorized, "unauthorized")
      }
      return next(c)
    }
  }
```

```
}
```

Mizu groups represent a router scope with an inherited middleware chain. The group carries a prefix and middleware that applies to routes registered on that group. Middleware composition uses a pure wrapping model where a middleware receives `next` and decides whether to call it, returning an error either way.

The technical payoff is a stable mental model: group composition behaves like structured handler wrapping, and the request path remains the real path rather than a rewritten one. That avoids the common StripPrefix confusion where internal handlers observe a different URL path from clients.

A quick comparison table helps anchor the implementation differences:

Framework	Group implemented as	Prefix handling	Middleware shape
net/http	mounted sub-mux	rewrite via StripPrefix	`func(next) handler`
Chi	scoped router view	internal prefix offset	`func(next) handler`
Gin	route builder	concatenated at registration	context chain, `Next/Abort`
Echo	scoped group + dispatcher	resolved by router	wrapper chain, returns error
Fiber	route builder	concatenated at registration	context chain, `Next`
Mizu	scoped router view	internal prefix + tree	wrapper chain, returns error

What learners should focus on

Groups look similar at the surface: a prefix and optional middleware. The deeper difference lies in where the composition happens and what the handler inside the group sees.

- net/http grouping mounts a handler under a prefix and often rewrites the path, which can affect redirects and logs inside the mounted handler
- Chi and Mizu keep the original path and implement groups as scoped router views with predictable middleware inheritance
- Gin and Fiber flatten prefixes and middleware at registration time and drive middleware flow through framework-controlled context progression
- Echo composes groups naturally with error returns, letting group middleware stop execution by returning an error and relying on centralized error handling

Middleware chaining and execution order

Middleware creates the request pipeline. Routing chooses a handler. Middleware decides what runs before the handler, what runs after it, and who gets to stop the request early. Once more than a few endpoints exist, most behavior lives in middleware: auth, logging, panic recovery, rate limiting, CORS, tracing, metrics, gzip, request IDs, and more. The important part sits underneath the API. Different stacks represent middleware differently, which changes how execution proceeds at runtime and how easy the order is to predict when the project grows.

This section focuses on three concrete questions:

- how middleware is represented in code
- how middleware is executed at runtime
- how a request stops or continues through the chain

The behavior stays intentionally small:

- one route: `GET /`
- two middleware layers
- each middleware prints a message before and after the handler

The observable print order reveals the true execution model. Each example below includes a link to the runnable code, the full `main.go`, and a walkthrough of how the pipeline is executed.

Expected output order when the request reaches the handler:

Stage	Print
1	`A before`
2	`B before`
3	`handler`
4	`B after`
5	`A after`

net/http

`nethttp/main.go`

```
package main

import (
  "fmt"
  "net/http"
)

func main() {
  handler := chain(
    http.HandlerFunc(finalHandler),
    middlewareA,
    middlewareB,
  )

  http.ListenAndServe(":8080", handler)
}

func finalHandler(w http.ResponseWriter, r *http.Request) {
```

```
  fmt.Fprintln(w, "handler")
}

type Middleware func(http.Handler) http.Handler

func middlewareA(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    fmt.Println("A before")
    next.ServeHTTP(w, r)
    fmt.Println("A after")
  })
}

func middlewareB(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    fmt.Println("B before")
    next.ServeHTTP(w, r)
    fmt.Println("B after")
  })
}

func chain(h http.Handler, m ...Middleware) http.Handler {
  for i := len(m) - 1; i >= 0; i-- {
    h = m[i](h)
  }
  return h
}
```

net/http middleware is plain functional composition. Each middleware takes a handler and returns a new handler. The chain is created by wrapping, which produces a nested call stack. The important part is that the order is locked in at build time, before any request arrives.

A small snippet captures the effective nesting:

```
h := middlewareA(middlewareB(http.HandlerFunc(finalHandler)))
```

When a request arrives, the server calls `h.ServeHTTP`. That enters middleware A. A prints its "before" line, then calls `next.ServeHTTP`, which enters middleware B. B prints its "before" line, then calls its next handler, which reaches the final handler. When the handler returns, execution unwinds back up the call stack, so B prints "after" and then A prints "after".

Early exit is a structural property. A middleware can stop the request by returning without calling `next.ServeHTTP`. Because the pipeline is a real call stack, "after" logic only runs for middleware that already called into the next handler.

Useful mental model:

Concept	net/http
Chain built	wrapper nesting
Continue	call `next.ServeHTTP`
Stop early	return without calling next
After code runs	only after next returns

Chi

chi/main.go

```
package main

import (
  "fmt"
  "net/http"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()

  r.Use(middlewareA)
  r.Use(middlewareB)

  r.Get("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "handler")
  })

  http.ListenAndServe(":8080", r)
}

func middlewareA(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    fmt.Println("A before")
    next.ServeHTTP(w, r)
    fmt.Println("A after")
  })
}

func middlewareB(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    fmt.Println("B before")
    next.ServeHTTP(w, r)
    fmt.Println("B after")
  })
}
```

Chi uses the same middleware type as net/http: `func(http.Handler) http.Handler`. The difference appears in where the chain is assembled. Middleware is registered on the router. At request time, after route matching selects an endpoint handler, Chi wraps that endpoint with the currently applicable middleware stack.

That means Chi delays the final chain composition until it knows which route matched. This enables scoping and inheritance for groups and subrouters. It also means a request to different routes can execute different middleware stacks, even though middleware is registered globally or in nested scopes.

Execution order still follows the wrapper call stack model. Middleware A wraps middleware B wraps the handler. Early exit works the same way as net/http: returning before calling next stops the chain.

A compact picture of what Chi effectively builds at dispatch time:

```
routeHandler := handlerFor("/")
h := middlewareA(middlewareB(routeHandler))
h.ServeHTTP(w, r)
```

Useful mental model:

Concept	Chi
Chain built	at dispatch time per route
Continue	call `next.ServeHTTP`
Stop early	return without calling next
After code runs	only after next returns

Gin

gin/main.go

```
package main

import (
  "fmt"
  "net/http"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()

  r.Use(middlewareA)
  r.Use(middlewareB)

  r.GET("/", func(c *gin.Context) {
    fmt.Fprintln(c.Writer, "handler")
```

```
  })

  r.Run(":8080")
}

func middlewareA(c *gin.Context) {
  fmt.Println("A before")
  c.Next()
  fmt.Println("A after")
}

func middlewareB(c *gin.Context) {
  fmt.Println("B before")
  c.Next()
  fmt.Println("B after")
}
```

Gin uses an index-driven execution model. Middleware and the final handler are stored together as a single ordered slice. The context holds an index into that slice. Calling `c.Next()` advances the index and runs subsequent handlers.

A useful way to think about it:

- the pipeline lives as data: `handlers []HandlerFunc`
- the call stack is simulated: `Next` moves forward and then returns back to the caller

The "before" and "after" pattern works because `c.Next()` blocks until the rest of the chain finishes. Once the handler returns, execution resumes at the line after `c.Next()`.

Early exit occurs when middleware does not call `c.Next()`, or when it aborts the chain. In Gin, aborting usually means setting a flag on the context that prevents further handlers from running.

A compact view of the state machine:

Step	Who runs	What advances
1	middlewareA	calls `Next`
2	middlewareB	calls `Next`
3	handler	returns
unwind	B resumes	then returns
unwind	A resumes	then returns

This model makes composition fast at request time because the chain is precomputed, but it introduces an execution state machine inside the context.

Echo

`echo/main.go`

```go
package main

import (
  "fmt"
  "net/http"

  "github.com/labstack/echo/v4"
)

func main() {
  e := echo.New()

  e.Use(middlewareA)
  e.Use(middlewareB)

  e.GET("/", func(c echo.Context) error {
    fmt.Fprintln(c.Response(), "handler")
    return nil
  })

  e.Start(":8080")
}

func middlewareA(next echo.HandlerFunc) echo.HandlerFunc {
  return func(c echo.Context) error {
    fmt.Println("A before")
    if err := next(c); err != nil {
      return err
    }
    fmt.Println("A after")
    return nil
  }
}

func middlewareB(next echo.HandlerFunc) echo.HandlerFunc {
  return func(c echo.Context) error {
    fmt.Println("B before")
    if err := next(c); err != nil {
      return err
    }
    fmt.Println("B after")
    return nil
  }
}
```

Echo uses wrapper composition like net/http, but the handler returns an error, and that error becomes the control channel. Each middleware receives `next` and returns a new handler. The chain forms a real call stack, and "after" logic runs when the call returns successfully.

The difference shows up when something fails. A middleware can stop the chain by returning an error. The dispatcher sees the error and routes it to centralized error handling. That gives a uniform failure path without context flags or abort states.

A minimal pattern for early exit in Echo middleware:

```
if !ok {
  return echo.NewHTTPError(http.StatusUnauthorized, "unauthorized")
}
return next(c)
```

This keeps stopping behavior explicit and testable as a plain return value.

Fiber

fiber/main.go

```
package main

import (
  "fmt"

  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()

  app.Use(middlewareA)
  app.Use(middlewareB)

  app.Get("/", func(c *fiber.Ctx) error {
    fmt.Println("handler")
    return c.SendString("handler")
  })

  app.Listen(":8080")
}

func middlewareA(c *fiber.Ctx) error {
  fmt.Println("A before")
  if err := c.Next(); err != nil {
    return err
  }
  fmt.Println("A after")
  return nil
}

func middlewareB(c *fiber.Ctx) error {
  fmt.Println("B before")
  if err := c.Next(); err != nil {
```

```
    return err
  }
  fmt.Println("B after")
  return nil
}
```

Fiber uses an index-driven chain like Gin, driven by `c.Next()`. Middleware and handlers are stored in an ordered slice, and the context advances through the slice.

The main difference from Gin is that error propagation is explicit. `c.Next()` returns an error, and returning a non-nil error stops execution and triggers the framework's error handling pipeline.

The mechanism remains a state machine inside the context. The context is pooled and reused, so the index and pipeline state must be reset per request. That makes correct lifecycle handling critical for correctness.

Mizu

`mizu/main.go`

```
package main

import (
  "fmt"

  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()

  app.Use(middlewareA)
  app.Use(middlewareB)

  app.Get("/", func(c *mizu.Ctx) error {
    fmt.Println("handler")
    return c.Text(200, "handler")
  })

  app.Listen(":8080")
}

func middlewareA(next mizu.Handler) mizu.Handler {
  return func(c *mizu.Ctx) error {
    fmt.Println("A before")
    if err := next(c); err != nil {
      return err
    }
    fmt.Println("A after")
    return nil
  }
}
```

```
func middlewareB(next mizu.Handler) mizu.Handler {
  return func(c *mizu.Ctx) error {
    fmt.Println("B before")
    if err := next(c); err != nil {
      return err
    }
    fmt.Println("B after")
    return nil
  }
}
```

Mizu middleware uses wrapper composition with explicit error returns. Each middleware receives `next` and returns a new handler. Execution follows a normal call stack: enter A, enter B, run handler, unwind back to B, unwind back to A.

Early exit is controlled by returning without calling next, or by returning an error. The error return provides a uniform failure channel that middleware can use to stop execution while still fitting clean composition.

The execution order stays visible in the code structure:

```
h := middlewareA(middlewareB(finalHandler))
```

What matters across stacks

Middleware models fall into two families:

Family	Frameworks	Mechanism	Continue	Stop
wrapping-based	net/http, Chi, Echo, Mizu	call stack via wrapper functions	call `next`	return early or return error
index-based	Gin, Fiber	state machine via context index	call `Next`	skip `Next`, abort, or return error

Both styles can produce the same observable order. The real difference appears when pipelines grow deep, when early exits become common, and when debugging execution order matters.

Short-circuiting and early exits

In real applications, many requests should never reach the final handler. Authentication failures, missing headers, invalid payloads, rate limits, feature flags, and maintenance windows all need a reliable way to stop execution early. Short-circuiting is the mechanism that enforces this stop.

The important detail is not whether a framework can stop a request. All of them can. The important detail is *how* the stop is enforced, *where* the signal lives, and *what guarantees exist* about what code may still run afterward.

This section uses a minimal setup:

- one route: `GET /`

- one middleware that rejects the request
- the final handler must never run

The examples show the full `main.go`, followed by a technical explanation of how execution is terminated inside the framework.

net/http

nethttp/main.go

```
package main

import (
  "net/http"
)

func main() {
  handler := deny(
    http.HandlerFunc(finalHandler),
  )

  http.ListenAndServe(":8080", handler)
}

func finalHandler(w http.ResponseWriter, r *http.Request) {
  w.Write([]byte("handler reached"))
}

func deny(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusUnauthorized)
    w.Write([]byte("denied"))
  })
}
```

Short-circuiting in net/http is a direct consequence of how middleware is composed. Middleware wraps a handler and decides whether to call it. The wrapper is the enforcement mechanism.

When the request arrives, the server calls the outermost handler. That handler writes a response and returns. Because it never calls `next.ServeHTTP`, no other handler is invoked. There is no signal, no flag, and no framework-managed state involved.

Once the middleware returns, the server considers the request complete and flushes the response. The final handler is unreachable by construction.

The guarantee is structural:

- execution proceeds only by calling `next.ServeHTTP`
- if that call never happens, nothing downstream can run

This property makes net/http short-circuiting easy to audit. The stop is visible in code and enforced by normal control flow.

Chi

chi/main.go

```
package main

import (
  "net/http"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()

  r.Use(deny)

  r.Get("/", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("handler reached"))
  })

  http.ListenAndServe(":8080", r)
}

func deny(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusUnauthorized)
    w.Write([]byte("denied"))
  })
}
```

Chi inherits net/http's wrapping model. Middleware is applied by wrapping handlers, and short-circuiting works the same way.

At request time, Chi resolves the route and builds a handler chain by wrapping the route handler with all applicable middleware. When the deny middleware runs, it writes a response and returns without calling `next.ServeHTTP`. The wrapped handler chain never progresses further.

No router state is modified. No abort mechanism exists. The stop is enforced by the absence of a function call.

The guarantee matches net/http:

- middleware controls execution by choosing whether to call next
- downstream handlers cannot run unless explicitly invoked

Gin

gin/main.go

```
package main

import (
  "net/http"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()

  r.Use(deny)

  r.GET("/", func(c *gin.Context) {
    c.String(http.StatusOK, "handler reached")
  })

  r.Run(":8080")
}

func deny(c *gin.Context) {
  c.AbortWithStatusJSON(http.StatusUnauthorized, gin.H{
    "error": "denied",
  })
}
```

Gin uses an index-based execution model. Middleware and handlers are stored in a single slice, and the context holds an index pointing to the current position in that slice.

When middleware runs, execution continues only if `c.Next()` is called. In this example, the middleware does not call `Next`. Instead, it calls `AbortWithStatusJSON`.

That call performs three actions:

- writes the response
- sets an internal aborted flag on the context
- prevents the index from advancing further

When control returns to the dispatcher, the aborted flag is checked and the remaining handlers are skipped.

The stop is not implicit. Writing a response alone does not stop execution. The framework enforces the stop only if the abort mechanism is triggered correctly.

The guarantee is conditional:

- execution stops only if `Abort` is called
- forgetting to abort allows execution to continue even after writing a response

This makes short-circuiting more powerful but also easier to misuse in security-sensitive middleware.

Echo

echo/main.go

```
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
)

func main() {
  e := echo.New()

  e.Use(deny)

  e.GET("/", func(c echo.Context) error {
    return c.String(http.StatusOK, "handler reached")
  })

  e.Start(":8080")
}

func deny(next echo.HandlerFunc) echo.HandlerFunc {
  return func(c echo.Context) error {
    return echo.NewHTTPError(http.StatusUnauthorized, "denied")
  }
}
```

Echo enforces short-circuiting through error returns. Middleware wraps the next handler and returns an error instead of calling it.

When the deny middleware returns an error, execution stops immediately. The dispatcher does not call the next handler. Instead, control transfers to the centralized error handler, which produces the response.

The stop signal is explicit and type-checked. Returning an error ends the chain. There is no separate abort flag and no reliance on side effects.

The guarantee is strong:

- once an error is returned, no further middleware or handlers run
- the error path is visible in the function signature

This model makes early exits easy to trace and test.

Fiber

fiber/main.go

```
package main
```

```
import (
  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()

  app.Use(deny)

  app.Get("/", func(c *fiber.Ctx) error {
    return c.SendString("handler reached")
  })

  app.Listen(":8080")
}

func deny(c *fiber.Ctx) error {
  return c.Status(401).SendString("denied")
}
```

Fiber also relies on error returns to stop execution, but the underlying execution model is index-based like Gin.

Middleware and handlers live in a slice. Calling `c.Next()` advances execution. In this example, `c.Next()` is never called. Instead, the middleware returns an error.

The dispatcher sees the error and terminates the request. Remaining handlers are skipped, and the response is sent.

Because Fiber contexts are pooled, the framework resets execution state after the request completes. The stop is enforced by the error return rather than by an abort flag.

The guarantee is clear:

- returning an error stops execution
- downstream handlers cannot run without `c.Next()`

Mizu

`mizu/main.go`

```
package main

import (
  "net/http"

  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()
```

```
  app.Use(deny)

  app.Get("/", func(c *mizu.Ctx) error {
    return c.Text(http.StatusOK, "handler reached")
  })

  app.Listen(":8080")
}

func deny(next mizu.Handler) mizu.Handler {
  return func(c *mizu.Ctx) error {
    return c.Text(http.StatusUnauthorized, "denied")
  }
}
```

Mizu uses wrapping-based middleware with an error-returning handler contract. The middleware decides whether to call `next`. In this example, it returns immediately with a response.

Because handlers return `error`, returning early naturally terminates execution. There is no abort flag and no index to manage. Execution stops by normal call stack unwinding.

The stop is enforced structurally:

- `next` is never called
- the returned error ends the chain
- downstream handlers remain unreachable

This combines the clarity of net/http wrapping with the explicit failure channel of Echo.

Comparing short-circuit behavior

Framework	Stop mechanism	How stop is enforced
net/http	do not call `next`	call stack structure
Chi	do not call `next`	call stack structure
Gin	`Abort*` methods	context flag + index
Echo	return error	dispatcher checks error
Fiber	return error	dispatcher checks error
Mizu	return error	call stack + error

The key difference lies in who enforces the stop.

- Wrapping-based models rely on normal control flow
- Index-based models require explicit signaling to the framework

This distinction matters when auditing authentication, authorization, and other critical middleware.

Error handling and panic recovery

Once a request can stop early, failure handling becomes the next hard requirement. Some failures are part of normal control flow: missing input, invalid state, rejected auth, conflict, rate limit. Other failures come from programmer mistakes or broken assumptions: nil pointer dereference, out-of-bounds slice access, double-close, unexpected type assertion, and deliberate `panic`.

This section separates those two failure modes:

- expected failures, represented as errors
- unexpected failures, represented as panics

The focus stays on mechanics: where a failure is observed, who turns it into an HTTP response, what runs after the failure, and what safety guarantees exist around response writing.

Scenario:

- `GET /error` produces an error response
- `GET /panic` panics
- the process stays alive and a response is produced

net/http

`nethttp/main.go`

```
package main

import (
  "fmt"
  "net/http"
)

func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("GET /error", errorHandler)
  mux.HandleFunc("GET /panic", panicHandler)

  handler := recoverMiddleware(mux)

  http.ListenAndServe(":8080", handler)
}

func errorHandler(w http.ResponseWriter, r *http.Request) {
  w.WriteHeader(http.StatusBadRequest)
  fmt.Fprintln(w, "bad request")
}

func panicHandler(w http.ResponseWriter, r *http.Request) {
```

```
  panic("something went wrong")
}

func recoverMiddleware(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    defer func() {
      if v := recover(); v != nil {
        w.WriteHeader(http.StatusInternalServerError)
        fmt.Fprintln(w, "internal server error")
      }
    }()
    next.ServeHTTP(w, r)
  })
}
```

net/http uses a write-to-response model for expected failures. A handler decides status code and body and writes them directly. That makes error handling entirely local to each handler unless a shared helper is introduced.

Panic recovery happens at the boundary around request execution. A panic unwinds the stack until a deferred function catches it with `recover`. The recovery wrapper then writes a 500 response.

The placement of the recovery wrapper determines what gets protected. Wrapping the mux protects handlers and anything inside the mux. Wrapping the entire server handler protects routing plus all middleware. The outermost wrapper becomes the last safety net.

One technical edge matters for correctness: response commitment. If a handler writes headers or body before panicking, a later recovery wrapper may be unable to change the status code to 500, because the response may already be committed.

A small guard pattern often appears in real servers:

```
// pseudo: if headers already sent, skip writing a second response
```

Core properties:

Failure type	How it surfaces	Who writes the response
expected failure	handler decides	handler
panic	recovered by wrapper	recovery wrapper

Chi

chi/main.go

```
package main

import (
  "fmt"
  "net/http"
```

```
  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()

  r.Use(recoverMiddleware)

  r.Get("/error", func(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusBadRequest)
    fmt.Fprintln(w, "bad request")
  })

  r.Get("/panic", func(w http.ResponseWriter, r *http.Request) {
    panic("something went wrong")
  })

  http.ListenAndServe(":8080", r)
}

func recoverMiddleware(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    defer func() {
      if v := recover(); v != nil {
        w.WriteHeader(http.StatusInternalServerError)
        fmt.Fprintln(w, "internal server error")
      }
    }()
    next.ServeHTTP(w, r)
  })
}
```

Chi keeps net/http semantics for errors and panics, then provides structure to attach cross-cutting behavior uniformly. Recovery is expressed as normal net/http middleware and applied at the router level, which makes it harder to forget in larger applications.

Expected failures still follow the same write-to-response pattern. A handler writes a 400 and returns.

Panic recovery behaves like the net/http wrapper model, with a key benefit: middleware placement is explicit and scoped. A router-level `Use` wraps route handlers consistently, including nested routes when groups are used.

Core properties:

Failure type	How it surfaces	Who writes the response
expected failure	handler decides	handler
panic	recovered by middleware	recovery middleware

Gin

gin/main.go

```
package main

import (
  "net/http"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()

  r.Use(gin.Recovery())

  r.GET("/error", func(c *gin.Context) {
    c.AbortWithStatusJSON(http.StatusBadRequest, gin.H{
      "error": "bad request",
    })
  })

  r.GET("/panic", func(c *gin.Context) {
    panic("something went wrong")
  })

  r.Run(":8080")
}
```

Gin separates two flows: expected failures flow through response-writing on the context, while panics flow through recovery middleware.

For expected failures, handlers use abort-style APIs. The abort both writes a response and signals the framework to stop the remaining chain.

For panics, `gin.Recovery()` wraps the handler chain. When a panic occurs, recovery catches it, logs stack information, and writes a 500 response.

The internal mechanism is tightly coupled to the context pipeline:

- middleware and handlers run in an ordered list
- abort changes control flow state inside the context
- recovery wraps execution so panics unwind into the recovery point

A practical detail shows up when combining abort and recovery: writing a response and then panicking later in the chain can create partially written responses. Recovery can only write a clean 500 response when the response has not been committed.

Core properties:

Failure type	How it surfaces	Who writes the response
expected failure	abort + write via context	handler
panic	recovered by middleware	recovery middleware

Echo

echo/main.go

```
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
  "github.com/labstack/echo/v4/middleware"
)

func main() {
  e := echo.New()

  e.Use(middleware.Recover())

  e.GET("/error", func(c echo.Context) error {
    return echo.NewHTTPError(http.StatusBadRequest, "bad request")
  })

  e.GET("/panic", func(c echo.Context) error {
    panic("something went wrong")
  })

  e.Start(":8080")
}
```

Echo routes expected failures through the handler return value. A handler returns an `error`, and the central dispatcher converts that error into an HTTP response.

Panic recovery is middleware. The recovery middleware catches the panic and turns it into an error that goes through the same centralized error handler.

This produces a single conversion point: the global error handler. That simplifies consistency. Status codes, error formatting, and logging can live in one place, while handlers can focus on returning meaningful errors.

A common pattern in larger Echo apps:

- middleware returns a typed HTTP error for expected failures
- unexpected panics become a generic 500 error through recovery
- the centralized error handler decides what to reveal to clients

Core properties:

Failure type	How it surfaces	Who writes the response
expected failure	returned error	centralized error handler
panic	recovered into error	centralized error handler

Fiber

fiber/main.go

```
package main

import (
  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New(fiber.Config{
    ErrorHandler: func(c *fiber.Ctx, err error) error {
      return c.Status(500).SendString("internal server error")
    },
  })

  app.Get("/error", func(c *fiber.Ctx) error {
    return fiber.NewError(400, "bad request")
  })

  app.Get("/panic", func(c *fiber.Ctx) error {
    panic("something went wrong")
  })

  app.Listen(":8080")
}
```

Fiber centralizes expected failures through the configured `ErrorHandler`. A handler returns an error, Fiber routes it to the global error handler, and the error handler writes the final response.

Panic recovery is handled within the request execution pipeline, converting the panic into an error that reaches the same error handler. This yields a single place for response formatting.

Because Fiber pools contexts, the framework must also guarantee cleanup after failures. The important property for users is consistency: expected errors and panics both flow toward the error handler, and the request completes with a response.

Core properties:

Failure type	How it surfaces	Who writes the response
expected failure	returned error	global error handler
panic	recovered into error	global error handler

Mizu

mizu/main.go

```
package main

import (
  "errors"
  "net/http"

  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()

  app.Get("/error", func(c *mizu.Ctx) error {
    return mizu.HTTPError{
      Status: http.StatusBadRequest,
      Err:    errors.New("bad request"),
    }
  })

  app.Get("/panic", func(c *mizu.Ctx) error {
    panic("something went wrong")
  })

  app.Listen(":8080")
}
```

Mizu routes failure through the handler error return. A handler returns an error value. When that error carries HTTP semantics, the framework converts it into a response with the matching status code. When the error carries no HTTP semantics, the framework converts it into a 500 response.

Panic recovery is part of the request execution pipeline. A panic is recovered, logged, and converted into an internal server error response. Expected failures and panics converge into the same conversion logic.

That convergence creates a stable rule: handlers focus on returning errors, and the framework focuses on response conversion and post-failure guarantees, including preventing process termination and keeping request teardown consistent.

Core properties:

Failure type	How it surfaces	Who writes the response
expected failure	returned error	centralized conversion
panic	recovered into error	centralized conversion

Comparing failure models

Framework	Expected failure path	Panic path	Central conversion point
net/http	handler writes response	outer recovery wrapper	optional, user-built
Chi	handler writes response	recovery middleware	optional, user-built
Gin	handler aborts and writes	recovery middleware	split between handler and recovery
Echo	handler returns error	recovered into error	global error handler
Fiber	handler returns error	recovered into error	configured error handler
Mizu	handler returns error	recovered into error	framework conversion pipeline

What to watch for when implementing real services:

- response commitment before failure, especially for panics after partial writes
- consistent status code mapping for domain errors
- consistent logging and client-visible messages for unexpected panics

Reading requests: headers, query, body

Request input is the boundary between the network and your code. A server parses bytes off a connection, builds a request object, and hands it to your handler. Everything after that depends on where the framework stores data, how it exposes it, and how it treats parsing errors. Headers and query parameters behave like metadata, while the body behaves like a stream. That difference shapes correctness more than any helper API.

This section uses the same logical requests for every framework:

- `GET /search?q=go`
- `POST /echo` with a JSON body

Focus points:

- where header and query data lives and how it is accessed
- when the body is consumed and whether it can be read twice
- where parse errors go and who turns them into HTTP responses
- what can be safely reused and what must be copied

net/http

`nethttp/main.go`

```
```

```go
package main

import (
  "encoding/json"
  "fmt"
  "net/http"
)

func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("GET /search", search)
  mux.HandleFunc("POST /echo", echo)

  http.ListenAndServe(":8080", mux)
}

func search(w http.ResponseWriter, r *http.Request) {
  q := r.URL.Query().Get("q")
  h := r.Header.Get("User-Agent")

  fmt.Fprintf(w, "query=%s ua=%s\n", q, h)
}

func echo(w http.ResponseWriter, r *http.Request) {
  defer r.Body.Close()

  var payload map[string]any
  if err := json.NewDecoder(r.Body).Decode(&payload); err != nil {
    w.WriteHeader(http.StatusBadRequest)
    w.Write([]byte("invalid json"))
    return
  }

  w.Header().Set("Content-Type", "application/json")
  json.NewEncoder(w).Encode(payload)
}
```

Headers and query parameters are already available by the time the handler runs. `r.Header` is a map-like structure holding header values, and `r.URL` contains the parsed URL, including query. Calling `r.URL.Query()` returns a parsed view of the query string. These are safe to read repeatedly because the parsing happens outside your handler and the data lives in memory attached to the request.

The body is different. `r.Body` is an `io.ReadCloser`, which represents a forward-only stream of bytes. JSON decoding consumes bytes from that stream. After `Decode` reads the stream, the data is gone unless it was buffered explicitly. A second decode attempt reads from the remaining bytes and typically returns `EOF` or produces incomplete input errors.

One subtle runtime detail: `json.Decoder` can leave trailing bytes unread if the input contains extra data. Many servers add strictness by decoding once and then verifying there is no trailing garbage. The base net/http approach leaves this decision to user code.

Parsing errors remain local. The decoder returns an error, and the handler decides status, body, logging, and whether to terminate.

Useful mental map:

Input	Where it lives	Read cost	Re-readable
headers	`r.Header`	cheap	yes
query	`r.URL` and `r.URL.Query()`	cheap	yes
body	`r.Body` stream	consumes bytes	no

Chi

`chi/main.go`

```
package main

import (
  "encoding/json"
  "fmt"
  "net/http"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()

  r.Get("/search", search)
  r.Post("/echo", echo)

  http.ListenAndServe(":8080", r)
}

func search(w http.ResponseWriter, r *http.Request) {
  q := r.URL.Query().Get("q")
  h := r.Header.Get("User-Agent")

  fmt.Fprintf(w, "query=%s ua=%s\n", q, h)
}

func echo(w http.ResponseWriter, r *http.Request) {
  defer r.Body.Close()

  var payload map[string]any
  if err := json.NewDecoder(r.Body).Decode(&payload); err != nil {
    w.WriteHeader(http.StatusBadRequest)
    w.Write([]byte("invalid json"))
    return
```

```
  }

  json.NewEncoder(w).Encode(payload)
}
```

Chi keeps request input semantics aligned with net/http. The handler receives `http.ResponseWriter` and `*http.Request`, and input parsing uses the same primitives. Chi does not introduce an alternative request object, so the same body-stream rules apply: decoding consumes the body.

The practical difference shows up when route params are involved. Chi stores params in the request context, but headers, query, and body remain standard. That makes it easier to reuse existing parsing libraries that expect `*http.Request`.

Error ownership stays local. Handler code decides how to map decode errors to responses unless an application-level wrapper is introduced.

Useful mental map:

Input	Access pattern
headers	`r.Header.Get`
query	`r.URL.Query().Get`
body	`json.NewDecoder(r.Body)`

Gin

`gin/main.go`

```
package main

import (
  "net/http"

  "github.com/gin-gonic/gin"
)

type Payload struct {
  Message string `json:"message"`
}

func main() {
  r := gin.New()

  r.GET("/search", func(c *gin.Context) {
    q := c.Query("q")
    ua := c.GetHeader("User-Agent")

    c.String(http.StatusOK, "query=%s ua=%s\n", q, ua)
  })
```

```
  r.POST("/echo", func(c *gin.Context) {
    var p Payload
    if err := c.BindJSON(&p); err != nil {
      c.AbortWithStatus(http.StatusBadRequest)
      return
    }
    c.JSON(http.StatusOK, p)
  })

  r.Run(":8080")
}
```

Gin exposes request input primarily through its context wrapper. Query and header helpers read from the underlying request but present a uniform API. That hides some details but keeps the common cases direct.

Body parsing happens through bind helpers like `BindJSON`. Under the hood, these helpers read from the request body stream and decode. The body still behaves as a one-shot stream: after it is consumed, the bytes are gone unless Gin is configured to buffer them in a specific mode.

A practical consequence is that calling `BindJSON` and later trying to read `c.Request.Body` again typically fails. In pipelines where multiple layers want access to the raw body, buffering must be introduced deliberately and early.

Error handling uses a hybrid approach. `BindJSON` returns an error, and user code decides whether to abort the chain. The framework does not automatically stop the request on bind failure. The abort call becomes the enforcement mechanism.

Useful mental map:

Input	Where it lives	Common accessor
headers	request + context helpers	`c.GetHeader`
query	request + context helpers	`c.Query`
body	request stream, decoded via helper	`c.BindJSON`
parse error	returned error + optional abort	`AbortWithStatus`

Echo

`echo/main.go`

```
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
)
```

```go
type Payload struct {
  Message string `json:"message"`
}

func main() {
  e := echo.New()

  e.GET("/search", func(c echo.Context) error {
    q := c.QueryParam("q")
    ua := c.Request().Header.Get("User-Agent")

    return c.String(http.StatusOK, "query="+q+" ua="+ua)
  })

  e.POST("/echo", func(c echo.Context) error {
    var p Payload
    if err := c.Bind(&p); err != nil {
      return echo.NewHTTPError(http.StatusBadRequest, "invalid json")
    }
    return c.JSON(http.StatusOK, p)
  })

  e.Start(":8080")
}
```

Echo combines direct access to the underlying request with convenience methods on its context. Query lookup is a direct name-based accessor. Headers can be read from `c.Request().Header` or via helpers depending on style.

Body parsing is performed by `Bind`, which consumes the body stream and decodes into a struct. The stream nature remains. Reading twice requires buffering.

The key difference lies in error flow. Handlers return `error`. Bind errors can be returned directly, which routes them into Echo's centralized error handling. That makes parse errors behave like first-class failures rather than local branches.

A small pattern helps keep parsing strict and readable:

- bind into struct
- return an HTTP error on failure
- proceed with validated struct

Useful mental map:

Input	Accessor	Failure flow
query	`c.QueryParam`	none
headers	`c.Request().Header.Get`	none
body	`c.Bind(&v)`	return error

Fiber

`fiber/main.go`

```
package main

import (
  "github.com/gofiber/fiber/v2"
)

type Payload struct {
  Message string `json:"message"`
}

func main() {
  app := fiber.New()

  app.Get("/search", func(c *fiber.Ctx) error {
    q := c.Query("q")
    ua := c.Get("User-Agent")

    return c.SendString("query=" + q + " ua=" + ua)
  })

  app.Post("/echo", func(c *fiber.Ctx) error {
    var p Payload
    if err := c.BodyParser(&p); err != nil {
      return c.Status(400).SendString("invalid json")
    }
    return c.JSON(p)
  })

  app.Listen(":8080")
}
```

Fiber is built on fasthttp. Headers and query values are accessed through the context, backed by fasthttp request structures. These accessors are fast and avoid allocations in many cases.

Body parsing is handled by `BodyParser`, which consumes the request body and decodes into a struct. The one-shot property remains. Multiple parses require buffering or storing the decoded value.

Because Fiber reuses contexts, request-scoped data must be treated as temporary. Values returned by accessors may reference internal buffers. Reading and immediately using values inside the handler is safe. Storing references beyond the handler requires copying.

Useful mental map:

Input	Accessor	Lifetime note
headers	`c.Get`	request-scoped
query	`c.Query`	request-scoped
body	`c.BodyParser`	consumes body

Mizu

`mizu/main.go`

```
package main

import (
  "net/http"

  "github.com/go-mizu/mizu"
)

type Payload struct {
  Message string `json:"message"`
}

func main() {
  app := mizu.New()

  app.Get("/search", func(c *mizu.Ctx) error {
    q := c.Query("q")
    ua := c.Request().Header.Get("User-Agent")

    return c.Text(http.StatusOK, "query="+q+" ua="+ua)
  })

  app.Post("/echo", func(c *mizu.Ctx) error {
    var p Payload
    if err := c.Bind(&p); err != nil {
      return err
    }
    return c.JSON(http.StatusOK, p)
  })

  app.Listen(":8080")
}
```

Mizu exposes the underlying net/http request while providing helpers for common patterns. Query can be read through helper methods. Headers remain available through the underlying request, which keeps compatibility with libraries that expect `*http.Request`.

Body parsing follows the stream rule. `Bind` consumes the body and decodes into a struct. A second bind reads no data unless buffering is added. Parse errors return as `error`, which flows into the framework's centralized error handling path, consistent with other failures.

This makes ownership clear:

- handler consumes input via explicit bind
- bind returns error on parse failure
- error return controls the response path

Useful mental map:

Input	Accessor	Failure flow
query	`c.Query`	none
headers	`c.Request().Header.Get`	none
body	`c.Bind(&v)`	return error

What to keep in mind

Across all frameworks, the body behaves like a stream. Reading consumes bytes. Helpers can hide the mechanics, but they do not change that fundamental property.

Useful checklist:

- headers and query are safe to read multiple times
- body must be treated as one-shot unless buffered
- parse errors determine control flow and error shape
- context reuse can affect lifetime of returned strings in some stacks

Writing responses: status, headers, streaming

Response output is stateful. Input can be inspected repeatedly, but output commits the connection to a specific status line, header set, and body bytes. After the first byte of the response is sent, the server cannot change its mind. That single constraint drives most of the differences between frameworks.

Three things happen during a response:

- metadata is assembled: status code and headers
- the first write commits headers and status to the wire
- body bytes are produced, either buffered and sent at once, or streamed in chunks

This section uses three patterns:

- a normal text response
- a JSON response with headers

- a streaming response that writes multiple chunks

The deep dive focuses on when headers become immutable, what “write twice” means in each stack, and what error handling can realistically do after output has started.

net/http

`nethttp/main.go`

```
package main

import (
  "encoding/json"
  "fmt"
  "net/http"
  "time"
)

func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("GET /text", text)
  mux.HandleFunc("GET /json", jsonResp)
  mux.HandleFunc("GET /stream", stream)

  http.ListenAndServe(":8080", mux)
}

func text(w http.ResponseWriter, r *http.Request) {
  w.WriteHeader(http.StatusOK)
  w.Write([]byte("hello"))
}

func jsonResp(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")
  w.WriteHeader(http.StatusOK)
  json.NewEncoder(w).Encode(map[string]string{
    "message": "hello",
  })
}

func stream(w http.ResponseWriter, r *http.Request) {
  flusher, ok := w.(http.Flusher)
  if !ok {
    http.Error(w, "streaming not supported", 500)
    return
  }

  for i := 0; i < 3; i++ {
    fmt.Fprintf(w, "chunk %d\n", i)
    flusher.Flush()
    time.Sleep(time.Second)
```

```
  }
}
```

In net/http, `http.ResponseWriter` is a live stream to the client. The server sends headers the moment the response is committed. Commitment happens on either of these events:

- an explicit `WriteHeader(status)`
- the first `Write([]byte(...))`, which implies `200 OK` if `WriteHeader` was never called

After commitment, status code cannot change and headers become effectively immutable. Setting headers after the first write has no effect because the header block has already been sent.

Two gotchas show up quickly:

- calling `WriteHeader` twice does not “update” the status; only the first call matters
- writing a partial body and then deciding an error happened can only affect logging, because the client already received part of the response

Streaming happens when the response is not fully buffered inside the server. `http.Flusher` exposes a mechanism to push buffered bytes down the connection before the handler returns. A handler that writes chunks and calls `Flush` can create observable partial progress on the client.

Response commitment model:

Action	What happens
set header before write	header will be sent
first `Write` without `WriteHeader`	status becomes 200, headers sent
`WriteHeader(500)` after first write	ignored
`Flush()`	pushes current buffered bytes

Chi

`chi/main.go`

```
package main

import (
  "encoding/json"
  "fmt"
  "net/http"
  "time"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()
```

```
  r.Get("/text", text)
  r.Get("/json", jsonResp)
  r.Get("/stream", stream)

  http.ListenAndServe(":8080", r)
}

func text(w http.ResponseWriter, r *http.Request) {
  w.Write([]byte("hello"))
}

func jsonResp(w http.ResponseWriter, r *http.Request) {
  w.Header().Set("Content-Type", "application/json")
  json.NewEncoder(w).Encode(map[string]string{
    "message": "hello",
  })
}

func stream(w http.ResponseWriter, r *http.Request) {
  flusher := w.(http.Flusher)

  for i := 0; i < 3; i++ {
    fmt.Fprintf(w, "chunk %d\n", i)
    flusher.Flush()
    time.Sleep(time.Second)
  }
}
```

Chi preserves net/http response semantics because the handler contract stays `http.ResponseWriter` and `*http.Request`. Every important rule remains the same: first write commits headers and status, and streaming uses `http.Flusher`.

The deep difference is not response output, but where response output can be wrapped. Since everything is a normal net/http handler, any buffering, compression, logging, or response recording layers can be introduced as middleware around the router.

One practical detail for streaming: the direct type assertion `w.(http.Flusher)` will panic if the writer does not support flushing. The net/http version used a safe check. That check matters when running under special writers such as test recorders.

Response model:

- same commit rules as net/http
- same streaming mechanism as net/http

Gin

`gin/main.go`

```
package main
```

```
import (
  "io"
  "net/http"
  "time"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()

  r.GET("/text", func(c *gin.Context) {
    c.String(http.StatusOK, "hello")
  })

  r.GET("/json", func(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{"message": "hello"})
  })

  r.GET("/stream", func(c *gin.Context) {
    c.Stream(func(w io.Writer) bool {
      for i := 0; i < 3; i++ {
        w.Write([]byte("chunk\n"))
        time.Sleep(time.Second)
      }
      return false
    })
  })

  r.Run(":8080")
}
```

Gin writes through its context. Helpers such as `String` and `JSON` coordinate three things in one call: status code, headers, and body encoding. Under the hood, Gin still sits on net/http, so headers ultimately commit when bytes are written to the underlying `ResponseWriter`.

The context acts as the policy surface. Multiple response helpers called in the same handler create an ordering problem: once the response has started, later attempts at setting status or headers are ineffective. Gin tracks response state and encourages a single “final response” path.

Streaming uses Gin’s `Stream` helper, which hands an `io.Writer` into a callback. The framework controls when headers are written and how the response is driven. This makes streaming feel structured, but it also means the streaming control flow is framework-owned, not a plain `Flusher` loop.

Commit behavior still follows the same physical rule: once bytes hit the underlying writer, headers are committed. The difference is that Gin tends to centralize response construction through helpers, which reduces accidental partial writes.

Execution expectations:

Pattern	Typical approach
text	`c.String(status, ...)`
JSON	`c.JSON(status, ...)`
streaming	`c.Stream(func(w io.Writer) bool { ... })`

Echo

`echo/main.go`

```
package main

import (
  "net/http"
  "time"

  "github.com/labstack/echo/v4"
)

func main() {
  e := echo.New()

  e.GET("/text", func(c echo.Context) error {
    return c.String(http.StatusOK, "hello")
  })

  e.GET("/json", func(c echo.Context) error {
    return c.JSON(http.StatusOK, map[string]string{"message": "hello"})
  })

  e.GET("/stream", func(c echo.Context) error {
    c.Response().Header().Set("Content-Type", "text/plain")
    for i := 0; i < 3; i++ {
      c.Response().Write([]byte("chunk\n"))
      c.Response().Flush()
      time.Sleep(time.Second)
    }
    return nil
  })

  e.Start(":8080")
}
```

Echo offers response helpers while still exposing the underlying response writer through `c.Response()`. That gives two styles:

- helper-driven: `c.String`, `c.JSON`
- writer-driven: `c.Response().Write`, `c.Response().Flush`

This is useful for streaming because streaming often needs control that high-level helpers deliberately abstract away.

Echo's error return affects response writing after commitment. If bytes have already been written and a handler returns an error, the framework can run the error handler, but it cannot reliably replace the response because the connection is already committed. In practice, once streaming starts, error returns become mostly a logging and cleanup channel.

Commit rules still derive from net/http. The main difference is that Echo makes "response already started" a practical concept because its centralized error pipeline depends on whether it still has the ability to write a fresh response.

Fiber

`fiber/main.go`

```
package main

import (
  "time"

  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()

  app.Get("/text", func(c *fiber.Ctx) error {
    return c.SendString("hello")
  })

  app.Get("/json", func(c *fiber.Ctx) error {
    return c.JSON(map[string]string{"message": "hello"})
  })

  app.Get("/stream", func(c *fiber.Ctx) error {
    c.Set("Content-Type", "text/plain")
    for i := 0; i < 3; i++ {
      c.WriteString("chunk\n")
      time.Sleep(time.Second)
    }
    return nil
  })

  app.Listen(":8080")
}
```

Fiber uses a fasthttp-based response model. The response is typically constructed in memory associated with the request context and written out when the handler returns. That makes "writing twice" mean "append to the response body buffer" rather than "send multiple independent chunks to a connection".

In this setup, repeated writes accumulate. Headers can usually be modified until the response is finalized. That changes the failure story: many errors can be handled late because nothing has been sent yet.

Streaming exists in Fiber, but it is not the default model. A plain loop writing strings does not guarantee immediate delivery to the client. The framework decides when to flush bytes to the network based on its own streaming APIs and transport behavior.

This makes typical API responses straightforward and efficient, while requiring explicit choices for real streaming semantics.

High-level behavior:

Operation	Typical behavior
multiple writes	append to body buffer
header changes late	often still effective
streaming loop	not guaranteed flush

Mizu

`mizu/main.go`

```go
package main

import (
  "net/http"
  "time"

  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()

  app.Get("/text", func(c *mizu.Ctx) error {
    return c.Text(http.StatusOK, "hello")
  })

  app.Get("/json", func(c *mizu.Ctx) error {
    return c.JSON(http.StatusOK, map[string]string{"message": "hello"})
  })

  app.Get("/stream", func(c *mizu.Ctx) error {
    c.SetHeader("Content-Type", "text/plain")
    for i := 0; i < 3; i++ {
      c.Write([]byte("chunk\n"))
      c.Flush()
      time.Sleep(time.Second)
    }
    return nil
```

```
  })

  app.Listen(":8080")
}
```

Mizu provides response helpers for the common cases while keeping low-level control available for streaming. `Text` and `JSON` establish status and write output in a single step. That reduces accidental "status set too late" situations and makes handler intent obvious.

For streaming, `Write` and `Flush` expose a net/http-like model: once the first write happens, headers are committed. Calling `Flush` pushes data through the underlying writer as chunks are produced.

The important constraint remains: after streaming starts, later errors cannot reliably change the response. An error returned after output begins becomes a signal for logging or middleware cleanup rather than a mechanism to generate a new HTTP error response.

Response commitment model:

- first write commits headers and status
- streaming requires explicit `Flush`
- error return after commitment cannot replace the response

What to keep in mind

Response semantics shape correctness:

- once the first byte is sent, status and headers are locked
- buffering makes late decisions possible, but changes streaming behavior
- streaming gives fine control, but makes mid-flight error recovery mostly impossible

A quick comparison:

Framework	Default model	Streaming style
net/http	direct to writer	`http.Flusher`
Chi	direct to writer	`http.Flusher`
Gin	helper-driven on writer	framework streaming helper
Echo	helper + writer access	manual write + flush
Fiber	buffered response build	explicit streaming APIs
Mizu	helpers + writer control	write + flush

JSON input and output

JSON APIs look simple on the surface: read bytes, decode into a struct, write bytes back. The hard parts start when a service grows and needs predictable failure behavior. JSON decoding consumes the request body stream, so the decision to decode in a helper, in middleware, or in the handler changes what later layers can do. The other pressure point is error shape: an invalid JSON body, a missing field, and an unexpected server failure all need to turn into consistent HTTP responses without duplicating logic across every handler.

This section keeps the scenarios small:

- decode JSON into a struct
- handle invalid JSON
- encode a JSON response

The focus stays on ownership and guarantees:

- where decoding happens
- when the body is consumed and whether it can be read again
- how errors propagate and where they become HTTP responses
- what validation means in practice and where it fits

net/http

`nethttp/main.go`

```
package main

import (
  "encoding/json"
  "net/http"
)

type Payload struct {
  Message string `json:"message"`
}

func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("POST /echo", func(w http.ResponseWriter, r *http.Request) {
    defer r.Body.Close()

    var p Payload
    if err := json.NewDecoder(r.Body).Decode(&p); err != nil {
      http.Error(w, "invalid json", http.StatusBadRequest)
      return
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(p)
  })
```

```
  http.ListenAndServe(":8080", mux)
}
```

Decoding is fully owned by the handler. The handler decides which decoder to use, when to close the body, and what happens on failure. The request body is an `io.Reader`, so decoding consumes it. A second decode attempt reads no bytes unless the handler buffered the body beforehand.

The default `json.Decoder` behavior matters for correctness. A typical API eventually wants two extra checks:

- reject unknown fields for schema stability
- reject trailing garbage after the first JSON value

A strict pattern often looks like this:

```
dec := json.NewDecoder(r.Body)
dec.DisallowUnknownFields()
if err := dec.Decode(&p); err != nil { /* 400 */ }
if dec.More() { /* 400 */ }
```

Encoding is symmetric but still manual. Header choice and status code choice remain the handler's responsibility. If encoding fails after headers have already been written, the handler cannot reliably change the status code. For JSON APIs, that pushes many teams toward writing through a buffer and only committing after encoding succeeds.

Ownership summary:

Step	Owner
decode	handler code
validation	handler or custom library
error mapping	handler code
encode	handler code

Chi

`chi/main.go`

```
package main

import (
  "encoding/json"
  "net/http"

  "github.com/go-chi/chi/v5"
)
```

```
type Payload struct {
  Message string `json:"message"`
}

func main() {
  r := chi.NewRouter()

  r.Post("/echo", func(w http.ResponseWriter, r *http.Request) {
    defer r.Body.Close()

    var p Payload
    if err := json.NewDecoder(r.Body).Decode(&p); err != nil {
      http.Error(w, "invalid json", http.StatusBadRequest)
      return
    }

    json.NewEncoder(w).Encode(p)
  })

  http.ListenAndServe(":8080", r)
}
```

Chi keeps the net/http JSON story unchanged. That means the same strengths and constraints: decoding is handler-owned, body consumption happens once, and failure mapping is a local decision unless a shared helper layer is introduced.

The net benefit is composability. Any decoding, strictness, or validation strategy built for `*http.Request` plugs in directly. Many services with Chi end up creating a small internal layer that standardizes “decode + validate + respond” and use it across endpoints.

A practical pattern looks like this:

- middleware sets request size limits and maybe a maximum body reader
- handler calls a shared decode helper
- helper returns typed errors that a shared error handler maps to JSON error responses

Ownership summary:

Concern	Result
JSON semantics	same as net/http
reuse outside framework	straightforward
centralized error format	app-defined

Gin

`gin/main.go`

```
package main
```

```
import (
  "net/http"

  "github.com/gin-gonic/gin"
)

type Payload struct {
  Message string `json:"message" binding:"required"`
}

func main() {
  r := gin.New()

  r.POST("/echo", func(c *gin.Context) {
    var p Payload
    if err := c.ShouldBindJSON(&p); err != nil {
      c.AbortWithStatusJSON(http.StatusBadRequest, gin.H{
        "error": err.Error(),
      })
      return
    }

    c.JSON(http.StatusOK, p)
  })

  r.Run(":8080")
}
```

Gin concentrates JSON decoding in a context method. `ShouldBindJSON` reads the body and decodes into the struct. The body remains a one-shot stream underneath; the helper does not change that fundamental constraint. After binding, later layers cannot re-read the raw body unless buffering was configured earlier.

The distinctive part is the validation integration. The struct tag `binding:"required"` participates in Gin's binding and validation pipeline. That makes "missing field" feel like a decode failure at the call site because both surface as an `err`. This reduces boilerplate but couples validation behavior to Gin's binding system and tag conventions.

Error propagation uses explicit control flow. The handler receives an error, decides to abort, and writes a response. Without aborting, middleware or later handlers can still run. That matters when teams build shared middleware that expects "bind failure stops everything" and forgets the abort.

A common stable pattern in Gin handlers:

```
if err := c.ShouldBindJSON(&p); err != nil {
  c.AbortWithStatusJSON(400, gin.H{"error": "bad request"})
  return
}
```

Ownership summary:

Step	Owner
decode	Gin binding
validation	Gin binding tags
error mapping	handler or shared middleware
stop execution	handler via abort

Echo

`echo/main.go`

```
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
)

type Payload struct {
  Message string `json:"message"`
}

func main() {
  e := echo.New()

  e.POST("/echo", func(c echo.Context) error {
    var p Payload
    if err := c.Bind(&p); err != nil {
      return echo.NewHTTPError(http.StatusBadRequest, "invalid json")
    }

    return c.JSON(http.StatusOK, p)
  })

  e.Start(":8080")
}
```

Echo's `Bind` consumes the body and decodes into the struct. The important difference is how failures flow. The handler returns an `error`, so bind failures can be returned directly and handled by the centralized error handler. That gives a natural place to standardize error responses without repeating response-writing logic across every handler.

Echo keeps validation separate from decoding by default. That separation can be useful: decoding answers "is it valid JSON", while validation answers "is it acceptable input". Many services implement validation via middleware or explicit calls after binding.

A common pattern:

```
if err := c.Bind(&p); err != nil { return echo.NewHTTPError(400, "invalid json") }
if p.Message == "" { return echo.NewHTTPError(400, "message required") }
return c.JSON(200, p)
```

Ownership summary:

Concern	Owner
decode	Echo bind
validation	app layer
error mapping	centralized handler
stop execution	returning error

Fiber

`fiber/main.go`

```
package main

import (
  "github.com/gofiber/fiber/v2"
)

type Payload struct {
  Message string `json:"message"`
}

func main() {
  app := fiber.New()

  app.Post("/echo", func(c *fiber.Ctx) error {
    var p Payload
    if err := c.BodyParser(&p); err != nil {
      return c.Status(400).SendString("invalid json")
    }
    return c.JSON(p)
  })

  app.Listen(":8080")
}
```

Fiber parses JSON through `BodyParser`. It consumes the body and fills the struct, returning an error on failure. Validation is typically layered above, similar to Echo, but the runtime model differs because Fiber is fasthttp-based and uses a pooled context.

For JSON response encoding, `c.JSON` handles encoding and content type. The response often behaves like a buffered build rather than immediate streaming, which makes it easier to commit a single consistent JSON response.

When building consistent API error shapes, teams typically standardize around a helper:

```
func badRequest(c *fiber.Ctx, msg string) error {
  return c.Status(400).JSON(fiber.Map{"error": msg})
}
```

Ownership summary:

Step	Owner
decode	Fiber parser
validation	app layer
error mapping	handler or shared helper
response encode	Fiber JSON helper

Mizu

`mizu/main.go`

```
package main

import (
  "net/http"

  "github.com/go-mizu/mizu"
)

type Payload struct {
  Message string `json:"message"`
}

func main() {
  app := mizu.New()

  app.Post("/echo", func(c *mizu.Ctx) error {
    var p Payload
    if err := c.Bind(&p); err != nil {
      return err
    }
    return c.JSON(http.StatusOK, p)
  })

  app.Listen(":8080")
}
```

Mizu makes JSON decoding an explicit operation that returns an error, then lets that error flow through the same error-handling path as other failures. The body is consumed during `Bind`, so the one-shot rule applies. The value is in consistency: decoding failures look like normal handler failures and can be handled centrally.

Encoding is explicit and symmetric via `c.JSON(status, value)`. That encourages a clean "parse input, validate input, return output" structure without mixing manual header setting and ad-hoc encoding in every handler.

A practical pattern is to keep validation separate but error-driven:

```
if err := c.Bind(&p); err != nil { return err }
if p.Message == "" { return mizu.HTTPError{Status: 400, Err: errors.New("message
required")} }
return c.JSON(200, p)
```

Ownership summary:

Concern	Owner
decode	Mizu bind
validation	app layer
error mapping	centralized error handling
encode	Mizu JSON helper

What to keep in mind

The main differences come from ownership and error flow:

- decoding in the handler gives maximum control but repeats logic
- decoding via context helpers reduces boilerplate but binds input parsing to framework types
- a returned-error handler model makes centralized error formatting easier
- validation can be coupled to binding tags or kept as an explicit app-layer step

A compact comparison:

Framework	Decode API	Validation	Error flow
net/http	manual decoder	manual	handler writes response
Chi	manual decoder	manual	handler writes response
Gin	bind helper	tag-driven optional	handler aborts + writes
Echo	bind helper	separate layer	return error
Fiber	body parser	separate layer	return error or response
Mizu	bind helper	separate layer	return error

Path parameters and typed access

Path parameters change routing from “does this string match” into “does this string match, and can the router hand me the interesting pieces cheaply and safely”. Once a pattern like `/users/:id` exists, the router performs extra work on every matching request: it must split or walk the path, decide which segments are variables, capture values, and expose them through a request-scoped API. That API then becomes the foundation for typed access. Every framework hands parameters back as strings, so typed access is really two parts: capture and conversion.

Capture has performance and correctness edges:

- capture must be request-scoped and safe under concurrency
- capture should avoid allocations when possible, but still be stable for the duration of the handler
- wildcard capture (`*path`) must decide whether the value includes a leading slash, and what happens when it captures nothing

Conversion has API design edges:

- conversion decides the error shape for malformed input
- conversion decides whether a missing parameter is a route mismatch or a handler-level error
- conversion determines how consistent failures are across the service

This section implements:

- `GET /users/:id` where `id` should parse as `int64`
- `GET /files/*path` where `path` is a wildcard segment

Then it discusses:

- route matches but `id` fails to parse
- route does not match
- wildcard capture yields an empty value

net/http

`nethttp/main.go`

```
package main

import (
  "fmt"
  "net/http"
  "strconv"
)

func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("GET /users/{id}", getUser)
  mux.HandleFunc("GET /files/{path...}", getFile)

  http.ListenAndServe(":8080", mux)
}

func getUser(w http.ResponseWriter, r *http.Request) {
  idStr := r.PathValue("id")

  id, err := strconv.ParseInt(idStr, 10, 64)
  if err != nil {
    w.WriteHeader(http.StatusBadRequest)
    fmt.Fprintf(w, "invalid id: %q\n", idStr)
    return
  }

  fmt.Fprintf(w, "user id=%d\n", id)
}

func getFile(w http.ResponseWriter, r *http.Request) {
  p := r.PathValue("path")
  fmt.Fprintf(w, "file path=%q\n", p)
}
```

`ServeMux` captures parameters as part of the pattern matcher. The capture storage is tied to the request so handlers read values through `r.PathValue(name)`. The important property: the mux owns capture, the handler owns conversion.

Conversion patterns usually fall into one of two styles:

- parse inline and write a 400 directly
- parse inline and return a typed error to a central error layer

A compact typed parse helper keeps handlers consistent:

```go
func pathInt64(r *http.Request, key string) (int64, error) {
  s := r.PathValue(key)
  if s == "" {
    return 0, fmt.Errorf("missing %s", key)
  }
  return strconv.ParseInt(s, 10, 64)
}
```

Wildcard `{path...}` captures the remainder of the path, including slashes. When the route matches `/files/`, the captured value can be empty depending on the exact incoming path. That makes `""` a valid capture and handlers should treat it intentionally.

Behavior focus:

Case	Result
`/users/123`	match, `PathValue("id") == "123"`
`/users/x`	match, parse fails, handler decides 400
`/users`	no match
`/files/a/b`	match, `PathValue("path") == "a/b"`
`/files/`	match, `PathValue("path")` may be `""`

Chi

`chi/main.go`

```go
package main

import (
  "fmt"
  "net/http"
  "strconv"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()

  r.Get("/users/{id}", getUser)
  r.Get("/files/*", getFile)

  http.ListenAndServe(":8080", r)
}

func getUser(w http.ResponseWriter, r *http.Request) {
  idStr := chi.URLParam(r, "id")
```

```
  id, err := strconv.ParseInt(idStr, 10, 64)
  if err != nil {
    w.WriteHeader(http.StatusBadRequest)
    fmt.Fprintf(w, "invalid id: %q\n", idStr)
    return
  }

  fmt.Fprintf(w, "user id=%d\n", id)
}

func getFile(w http.ResponseWriter, r *http.Request) {
  p := chi.URLParam(r, "*")
  fmt.Fprintf(w, "file path=%q\n", p)
}
```

Chi captures parameters during route matching and stores them in routing state attached to the request context. Retrieval via `chi.URLParam` keeps handler signatures net/http-shaped, but the lookup is router-defined.

Typed conversion stays handler-owned. For consistency, many Chi codebases wrap conversion in helpers that return `(T, bool)` or `(T, error)` and centralize 400 formatting.

Wildcard capture uses `/*` and is retrieved as `"*"`. That special key becomes part of the service's conventions. It helps to normalize it immediately:

```
p := chi.URLParam(r, "*")
if p == "" { /* empty capture path */ }
```

Chi routes do not rewrite the request path, so captured values correspond to the real incoming URL path. That makes logs and redirects consistent without extra work.

Gin

`gin/main.go`

```
package main

import (
  "net/http"
  "strconv"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()

  r.GET("/users/:id", func(c *gin.Context) {
    idStr := c.Param("id")
```

```
    id, err := strconv.ParseInt(idStr, 10, 64)
    if err != nil {
      c.AbortWithStatusJSON(http.StatusBadRequest, gin.H{
        "error": "invalid id",
        "id":    idStr,
      })
      return
    }

    c.JSON(http.StatusOK, gin.H{"id": id})
  })

  r.GET("/files/*path", func(c *gin.Context) {
    // Gin includes the leading slash in wildcard params by default.
    p := c.Param("path")
    c.String(http.StatusOK, "file path=%q", p)
  })

  r.Run(":8080")
}
```

Gin captures params in the router and exposes them through the context. Retrieval stays string-based, so typed access is always an explicit parse step.

The important internal behavior difference is wildcard normalization. Gin's `*path` typically includes a leading slash in the captured value. That affects code that wants to join paths safely. A stable pattern trims it once:

```
p := c.Param("path")
if len(p) > 0 && p[0] == '/' {
  p = p[1:]
}
```

Conversion failures typically stop the request through abort semantics rather than an error return. That makes "typed parameter parsing" part of the request-control policy of the service.

Behavior focus:

Case	Typical handler choice
parse fails	`AbortWithStatusJSON(400, ...)`
wildcard value	normalize leading slash before use

Echo

`echo/main.go`

```
package main
```

```
import (
  "net/http"
  "strconv"

  "github.com/labstack/echo/v4"
)

func main() {
  e := echo.New()

  e.GET("/users/:id", func(c echo.Context) error {
    idStr := c.Param("id")

    id, err := strconv.ParseInt(idStr, 10, 64)
    if err != nil {
      return echo.NewHTTPError(http.StatusBadRequest, "invalid id")
    }

    return c.JSON(http.StatusOK, map[string]any{"id": id})
  })

  e.GET("/files/*", func(c echo.Context) error {
    // Echo exposes wildcard captures with Param("*")
    p := c.Param("*")
    return c.String(http.StatusOK, "file path="+p)
  })

  e.Start(":8080")
}
```

Echo stores route params on its context, and `Param(name)` retrieves them. Typed parsing remains explicit. The main difference is the error model: parse failure can return an error value, which flows into centralized error handling.

That error return path makes typed access patterns easier to standardize. A helper that returns `(int64, error)` composes naturally:

```
func paramInt64(c echo.Context, key string) (int64, error) {
  s := c.Param(key)
  id, err := strconv.ParseInt(s, 10, 64)
  if err != nil {
    return 0, echo.NewHTTPError(400, "invalid "+key)
  }
  return id, nil
}
```

Wildcard capture uses `*` and is retrieved as `Param("*")`. Empty capture should be treated intentionally as a valid match for routes like `/files/`.

Fiber

`fiber/main.go`

```
package main

import (
  "strconv"

  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()

  app.Get("/users/:id", func(c *fiber.Ctx) error {
    idStr := c.Params("id")

    id, err := strconv.ParseInt(idStr, 10, 64)
    if err != nil {
      return c.Status(400).SendString("invalid id")
    }

    return c.JSON(map[string]any{"id": id})
  })

  app.Get("/files/*", func(c *fiber.Ctx) error {
    // Fiber wildcard is Params("*")
    p := c.Params("*")
    return c.SendString("file path=" + p)
  })

  app.Listen(":8080")
}
```

Fiber captures params during match and stores them in the request context object, which is pooled. The handler reads params as strings via `Params(name)`.

For typed access, the main practical rule is to treat param values as request-scoped data. Parse immediately into a typed value and store the typed result if it needs to move deeper into the application.

Wildcard capture is retrieved through `Params("*")`. Empty capture can happen for `/files/` and should be treated as either a valid “root of files” request or an input error, depending on service policy.

Mizu

`mizu/main.go`

```
package main

import (
  "net/http"
  "strconv"
```

```
  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()

  app.Get("/users/:id", func(c *mizu.Ctx) error {
    idStr := c.Param("id")

    id, err := strconv.ParseInt(idStr, 10, 64)
    if err != nil {
      return c.Text(http.StatusBadRequest, "invalid id")
    }

    return c.JSON(http.StatusOK, map[string]any{"id": id})
  })

  app.Get("/files/*path", func(c *mizu.Ctx) error {
    p := c.Param("path")
    return c.Text(http.StatusOK, "file path="+p)
  })

  app.Listen(":8080")
}
```

Mizu captures params during routing and stores them on the request-scoped context. Retrieval via `c.Param(name)` keeps handlers framework-shaped while keeping the conversion step explicit and visible.

Typed access is usually structured as a small helper that returns `(T, error)` and plugs into the handler's error-return model:

```
func paramInt64(c *mizu.Ctx, key string) (int64, error) {
  s := c.Param(key)
  id, err := strconv.ParseInt(s, 10, 64)
  if err != nil {
    return 0, mizu.HTTPError{Status: 400, Err: err}
  }
  return id, nil
}
```

Wildcard capture uses a named star segment `*path`, which avoids the `"*"` magic key pattern and makes code more self-documenting. Empty capture should be treated explicitly, especially for `/files/` style routes.

What to pay attention to

Every stack follows the same broad shape: route matches, router captures strings, handler parses to types. The real differences show up in the edges:

- where the captured values live (request context, router context, pooled context)

- how wildcards are represented and whether the captured value includes a leading slash
- how parse failures stop execution and how error responses are formed

A compact comparison:

Framework	Param API	Wildcard API	Failure style
net/http	`r.PathValue("id")`	`{path...}`	handler writes response
Chi	`chi.URLParam(r,"id")`	key `"*"`	handler writes response
Gin	`c.Param("id")`	`*path` often includes leading `/`	abort + write response
Echo	`c.Param("id")`	`Param("*")`	return error
Fiber	`c.Params("id")`	`Params("*")`	write response / return
Mizu	`c.Param("id")`	`*path` named	return error or response

Static files and embedded assets

Serving static content looks straightforward, but the moment a real system is involved, several low level rules come into play. A request for a file is still an HTTP request, which means routing, path normalization, headers, streaming, and error handling all apply. Static serving also becomes the first place where deployment choices matter, especially when assets move from disk into the binary.

This section focuses on what actually happens when a file is requested and how each framework wires that behavior. Two cases are covered throughout:

- serving files from disk
- serving files embedded into the binary

The intent is to understand ownership and data flow, not to memorize helpers.

net/http

```
package main

import (
  "embed"
  "net/http"
)

//go:embed public/*
var assets embed.FS

func main() {
  mux := http.NewServeMux()

  // serve from disk
  mux.Handle("/static/",
```

```
    http.StripPrefix("/static/",
      http.FileServer(http.Dir("./public")),
    ),
  )

  // serve embedded files
  fs := http.FS(assets)
  mux.Handle("/embed/",
    http.StripPrefix("/embed/",
      http.FileServer(fs),
    ),
  )

  http.ListenAndServe(":8080", mux)
}
```

How static serving works

`http.FileServer` is just another `http.Handler`. It takes the request path, cleans it, maps it onto a filesystem rooted at a directory or an `fs.FS`, opens the file, sets headers such as `Content-Type`, `Last-Modified`, and `Content-Length`, then streams the file to the client.

The handler does not know about routing prefixes. That is why `http.StripPrefix` is required. Without stripping `/static` or `/embed`, the file server would attempt to open paths like `./public/static/app.css`, which usually do not exist.

Embedded files work because `http.FS` adapts any `fs.FS` into the interface expected by `FileServer`. From the file server's point of view, disk and embedded files behave the same. The difference is only in how bytes are retrieved.

Security is handled by path cleaning inside `FileServer`. Directory traversal attempts like `../` are rejected as long as the filesystem root is correctly defined.

Chi

```
package main

import (
  "embed"
  "net/http"

  "github.com/go-chi/chi/v5"
)

//go:embed public/*
var assets embed.FS

func main() {
  r := chi.NewRouter()

  r.Handle("/static/*",
```

```
    http.StripPrefix("/static/",
      http.FileServer(http.Dir("./public")),
    ),
  )

  r.Handle("/embed/*",
    http.StripPrefix("/embed/",
      http.FileServer(http.FS(assets)),
    ),
  )

  http.ListenAndServe(":8080", r)
}
```

How static serving works

Chi does not implement static serving itself. Routing decides which handler runs. Once the request reaches the file server handler, everything is standard library behavior.

The important detail is that Chi does not rewrite request paths internally. Prefix stripping remains an explicit step. That makes static serving predictable and consistent with other handlers.

Because the file server is a normal handler, it participates naturally in middleware chains. Authentication, logging, and compression can wrap static content without special cases.

Gin

```
package main

import (
  "embed"

  "github.com/gin-gonic/gin"
)

//go:embed public/*
var assets embed.FS

func main() {
  r := gin.New()

  // serve from disk
  r.Static("/static", "./public")

  // serve embedded files
  r.StaticFS("/embed", gin.FS(assets))

  r.Run(":8080")
}
```

How static serving works

Gin exposes static serving through helpers. `Static` and `StaticFS` register routes and internally construct file serving handlers.

Prefix handling, content type detection, and common headers are configured for you. This reduces boilerplate, but it also hides the fact that the underlying mechanism is still a file server handler.

Files are streamed rather than fully loaded into memory. The handler decides when headers are sent, and the response behaves like any other Gin response once writing begins.

Because static serving is integrated at the router level, it follows Gin's execution and abort rules.

Echo

```
package main

import (
  "embed"

  "github.com/labstack/echo/v4"
)

//go:embed public/*
var assets embed.FS

func main() {
  e := echo.New()

  e.Static("/static", "public")
  e.StaticFS("/embed", assets)

  e.Start(":8080")
}
```

How static serving works

Echo provides first class helpers for static files. These helpers configure internal handlers that map request paths to filesystem paths and stream file contents.

Static serving is integrated with Echo's middleware and error handling pipeline. A missing file typically results in a 404 response handled by the framework rather than a raw handler write.

Because Echo handlers return errors, static serving failures flow through the same centralized error logic as application handlers.

Fiber

```
package main

import (
  "embed"
```

```
  "github.com/gofiber/fiber/v2"
)

//go:embed public/*
var assets embed.FS

func main() {
  app := fiber.New()

  app.Static("/static", "./public")
  app.Static("/embed", "./public") // Fiber does not support embed.FS directly

  app.Listen(":8080")
}
```

How static serving works

Fiber uses fasthttp based file serving utilities. Static handlers are optimized for speed and often buffer more aggressively than net/http based servers.

Direct support for `embed.FS` is limited. Many Fiber applications either avoid embedding static assets or copy embedded files to disk during startup.

This limitation follows from Fiber's non net/http foundation. The tradeoff favors performance and simplicity over compatibility with standard library abstractions.

Mizu

```
package main

import (
  "embed"
  "net/http"

  "github.com/go-mizu/mizu"
)

//go:embed public/*
var assets embed.FS

func main() {
  app := mizu.New()

  app.Static("/static", "./public")
  app.StaticFS("/embed", http.FS(assets))

  app.Listen(":8080")
}
```

How static serving works

Mizu exposes static serving explicitly while keeping net/http semantics intact.

`Static` serves files from disk. `StaticFS` serves files from any `fs.FS`, including embedded assets. Internally, Mizu constructs handlers using standard file serving logic and integrates them into its routing and middleware pipeline.

Because static handlers are ordinary handlers, middleware such as logging, authentication, or rate limiting applies consistently. Embedded and disk based assets behave the same at request time.

What to focus on

Static serving reveals how much a framework hides or exposes.

Important differences to pay attention to:

- whether path rewriting is explicit or implicit
- whether embedded files are first class
- whether responses are streamed or buffered
- how missing files are translated into HTTP responses

These details influence security posture, memory usage, and how easily assets move between development and deployment.

Templates and HTML rendering

Templates introduce server side rendering and response composition. Instead of returning raw bytes or structured data, the server now produces HTML that combines static markup with dynamic data. This shifts responsibility from the client to the server and raises new questions about ownership, lifecycle, and failure modes.

At this point in the stack, several technical details start to matter:

- when templates are parsed and cached
- how data is passed into templates
- whether output is streamed directly or buffered first
- how layouts and partials are composed
- what happens when rendering fails after writing has started

All examples render a simple page with a title and a message. The differences lie in how rendering is wired and how much control the framework takes.

net/http

```
package main

import (
  "html/template"
```

```
  "net/http"
)

var tpl = template.Must(template.New("page").Parse(`
<!doctype html>
<html>
<head><title>{{.Title}}</title></head>
<body>
<h1>{{.Message}}</h1>
</body>
</html>
`))

type Data struct {
  Title   string
  Message string
}

func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("GET /", func(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "text/html; charset=utf-8")
    if err := tpl.Execute(w, Data{
      Title:   "Home",
      Message: "hello from template",
    }); err != nil {
      http.Error(w, "template error", http.StatusInternalServerError)
    }
  })

  http.ListenAndServe(":8080", mux)
}
```

How template rendering works

Templates are parsed once at program startup. The parsed template is a Go value that can be executed many times concurrently. Parsing errors fail fast at startup rather than at request time.

When a request arrives, the handler calls `Execute` with the response writer and a data value. Template execution writes directly to the writer. This means output is streamed as the template runs.

This streaming behavior has an important consequence. If execution fails after some bytes have already been written, the response status and headers are already committed. At that point, the handler can no longer change the status code or recover cleanly. Many production systems address this by rendering into a buffer first, then copying the result to the response only if rendering succeeds.

The standard library gives full control over parsing, execution, and error handling, but it also makes all tradeoffs explicit.

Chi

```go
package main

import (
  "html/template"
  "net/http"

  "github.com/go-chi/chi/v5"
)

var tpl = template.Must(template.New("page").Parse(`
<h1>{{.Message}}</h1>
`))

func main() {
  r := chi.NewRouter()

  r.Get("/", func(w http.ResponseWriter, r *http.Request) {
    tpl.Execute(w, map[string]string{
      "Message": "hello from chi",
    })
  })

  http.ListenAndServe(":8080", r)
}
```

How template rendering works

Chi does not introduce a rendering abstraction. The handler executes templates exactly as in net/http.

Routing determines which handler runs, but rendering remains an ordinary function call that writes to the response writer. Parsing strategy, buffering, layout composition, and error handling are all application concerns.

This makes Chi a good fit for teams that already have a rendering layer or want to share rendering logic outside HTTP handlers.

Gin

```go
package main

import (
  "net/http"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()

  r.LoadHTMLGlob("templates/*")
```

```
  r.GET("/", func(c *gin.Context) {
    c.HTML(http.StatusOK, "page.html", gin.H{
      "title":   "Home",
      "message": "hello from gin",
    })
  })

  r.Run(":8080")
}
```

How template rendering works

Gin integrates template rendering into the framework lifecycle.

Templates are parsed during startup when `LoadHTMLGlob` is called and stored on the engine. Layouts and partials are supported through standard Go template definitions.

Calling `c.HTML` sets the status code, headers, and renders the template. Output is buffered internally. Because of buffering, Gin can detect rendering errors before sending headers and still return an appropriate status code.

This approach reduces boilerplate and avoids partial responses, but it also ties rendering configuration to the Gin engine. Rendering is no longer a plain function call independent of the framework.

Echo

```
package main

import (
  "html/template"
  "net/http"

  "github.com/labstack/echo/v4"
)

type TemplateRenderer struct {
  t *template.Template
}

func (r *TemplateRenderer) Render(w http.ResponseWriter, name string, data any, c
echo.Context) error {
  return r.t.ExecuteTemplate(w, name, data)
}

func main() {
  e := echo.New()

  e.Renderer = &TemplateRenderer{
    t: template.Must(template.ParseGlob("templates/*.html")),
  }

  e.GET("/", func(c echo.Context) error {
```

```
    return c.Render(http.StatusOK, "page.html", map[string]string{
      "Message": "hello from echo",
    })
  })

  e.Start(":8080")
}
```

How template rendering works

Echo makes rendering explicit by requiring a renderer interface.

The framework does not assume a specific template engine. Instead, the application provides a renderer that knows how to render templates. This renderer is responsible for parsing, execution, and error behavior.

Handlers call `c.Render`, which delegates to the configured renderer. Errors returned from rendering propagate through Echo's centralized error handling.

This design makes rendering pluggable and explicit, at the cost of a small amount of setup code.

Fiber

```
package main

import (
  "github.com/gofiber/fiber/v2"
  "github.com/gofiber/template/html/v2"
)

func main() {
  engine := html.New("./templates", ".html")

  app := fiber.New(fiber.Config{
    Views: engine,
  })

  app.Get("/", func(c *fiber.Ctx) error {
    return c.Render("page", fiber.Map{
      "Message": "hello from fiber",
    })
  })

  app.Listen(":8080")
}
```

How template rendering works

Fiber treats templates as a view engine configured at application creation time.

The engine parses templates and handles rendering. Handlers call `Render`, which produces output and returns an error.

Rendering is buffered. Headers are not sent until rendering completes successfully. This avoids partial responses and makes error handling straightforward.

Template engines live in external packages rather than the core. This keeps the core small but requires choosing and configuring a renderer explicitly.

Mizu

```
package main

import (
  "html/template"
  "net/http"

  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()

  tpl := template.Must(template.ParseGlob("templates/*.html"))

  app.Get("/", func(c *mizu.Ctx) error {
    c.SetHeader("Content-Type", "text/html; charset=utf-8")
    return tpl.Execute(c.Writer(), map[string]string{
      "Message": "hello from mizu",
    })
  })

  app.Listen(":8080")
}
```

How template rendering works

Mizu does not impose a rendering abstraction.

Templates are standard Go templates. Execution is explicit and returns an error. The handler decides how to handle failures and whether to buffer output.

Because Mizu exposes the underlying writer, streaming and incremental rendering are possible. This provides flexibility, but it also requires care when handling errors, since headers may already be sent.

Rendering fits naturally into Mizu's error-return handler model, but control remains with the application.

What to focus on

Template rendering highlights how much a framework wants to manage for you.

Important questions to keep in mind:

- whether rendering is streamed or buffered
- whether templates are tied to the router or standalone

- whether errors are handled locally or centrally
- whether rendering logic can be reused outside HTTP handlers

These decisions influence performance, failure behavior, and how easily rendering logic evolves over time.

Forms, multipart data, and file uploads

Forms and file uploads exercise parts of HTTP that JSON APIs often bypass. They combine multiple concerns in a single request: structured fields, streamed bodies, temporary storage, size limits, and cleanup rules. They also introduce real risk if handled casually, because a single request can allocate large amounts of memory or disk space.

At this level, frameworks must answer several concrete questions:

- when the request body is parsed
- where parsed form fields live
- where uploaded files are stored
- who is responsible for cleanup
- how size limits are enforced
- what happens if parsing fails halfway through

The examples below implement two endpoints:

- `POST /login` using form fields
- `POST /upload` using `multipart/form-data` with a file field named `file`

The focus is not convenience, but understanding data ownership and lifecycle.

net/http

```
package main

import (
  "fmt"
  "io"
  "net/http"
  "os"
)

func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("POST /login", login)
  mux.HandleFunc("POST /upload", upload)

  http.ListenAndServe(":8080", mux)
}
```

```
func login(w http.ResponseWriter, r *http.Request) {
  if err := r.ParseForm(); err != nil {
    http.Error(w, "bad form", http.StatusBadRequest)
    return
  }

  user := r.FormValue("user")
  pass := r.FormValue("pass")

  fmt.Fprintf(w, "user=%s pass=%s\n", user, pass)
}

func upload(w http.ResponseWriter, r *http.Request) {
  if err := r.ParseMultipartForm(10 << 20); err != nil {
    http.Error(w, "bad multipart", http.StatusBadRequest)
    return
  }
  defer r.MultipartForm.RemoveAll()

  file, header, err := r.FormFile("file")
  if err != nil {
    http.Error(w, "file missing", http.StatusBadRequest)
    return
  }
  defer file.Close()

  out, err := os.Create("./" + header.Filename)
  if err != nil {
    http.Error(w, "cannot save file", http.StatusInternalServerError)
    return
  }
  defer out.Close()

  io.Copy(out, file)

  fmt.Fprintf(w, "uploaded %s\n", header.Filename)
}
```

How form and upload handling works

In net/http, form parsing is explicit and destructive. Calling `ParseForm` or `ParseMultipartForm` consumes the request body and populates internal data structures on the request.

For multipart requests, the standard library automatically spills large parts to disk once a memory threshold is exceeded. The `maxMemory` argument controls how much data is kept in memory before temporary files are created.

Temporary files are not cleaned up automatically. Calling `RemoveAll` on `r.MultipartForm` is required to avoid leaking disk space.

The handler owns everything: limits, validation, storage location, and cleanup. This provides maximum control, but also means every mistake is yours.

Chi

```
package main

import (
  "fmt"
  "io"
  "net/http"
  "os"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()

  r.Post("/login", login)
  r.Post("/upload", upload)

  http.ListenAndServe(":8080", r)
}

func login(w http.ResponseWriter, r *http.Request) {
  r.ParseForm()
  fmt.Fprintf(w, "user=%s\n", r.FormValue("user"))
}

func upload(w http.ResponseWriter, r *http.Request) {
  r.ParseMultipartForm(10 << 20)
  defer r.MultipartForm.RemoveAll()

  file, header, _ := r.FormFile("file")
  defer file.Close()

  out, _ := os.Create(header.Filename)
  defer out.Close()

  io.Copy(out, file)

  fmt.Fprintf(w, "uploaded %s\n", header.Filename)
}
```

How form and upload handling works

Chi does not modify form or multipart semantics. Everything behaves exactly as in net/http.

This means:

- the body is consumed when parsing functions are called

- temporary files may be created automatically
- cleanup remains the handler's responsibility

Chi adds routing structure, but form handling remains a standard library concern.

Gin

```
package main

import (
  "net/http"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()

  r.POST("/login", func(c *gin.Context) {
    user := c.PostForm("user")
    pass := c.PostForm("pass")

    c.String(http.StatusOK, "user=%s pass=%s", user, pass)
  })

  r.POST("/upload", func(c *gin.Context) {
    file, err := c.FormFile("file")
    if err != nil {
      c.AbortWithStatus(http.StatusBadRequest)
      return
    }

    c.SaveUploadedFile(file, "./"+file.Filename)
    c.String(http.StatusOK, "uploaded %s", file.Filename)
  })

  r.Run(":8080")
}
```

How form and upload handling works

Gin parses form and multipart data lazily. Parsing happens when helpers such as `PostForm` or `FormFile` are first called.

Uploaded files are represented by headers and temporary storage managed by the framework. `SaveUploadedFile` copies the uploaded content to a destination path and abstracts away file handling details.

This reduces boilerplate, but it also hides:

- where temporary files are stored

- when cleanup happens
- what size limits are enforced

Large uploads still require configuring limits at the server or reverse proxy level.

Echo

```
package main

import (
  "io"
  "net/http"
  "os"

  "github.com/labstack/echo/v4"
)

func main() {
  e := echo.New()

  e.POST("/login", func(c echo.Context) error {
    user := c.FormValue("user")
    pass := c.FormValue("pass")

    return c.String(http.StatusOK, "user="+user+" pass="+pass)
  })

  e.POST("/upload", func(c echo.Context) error {
    file, err := c.FormFile("file")
    if err != nil {
      return echo.NewHTTPError(http.StatusBadRequest)
    }

    src, err := file.Open()
    if err != nil {
      return err
    }
    defer src.Close()

    dst, err := os.Create(file.Filename)
    if err != nil {
      return err
    }
    defer dst.Close()

    io.Copy(dst, src)

    return c.String(http.StatusOK, "uploaded "+file.Filename)
  })

  e.Start(":8080")
```

```
}
```

How form and upload handling works

Echo exposes form values through context helpers, but file handling remains explicit.

Multipart parsing and temporary file storage are handled by the underlying net/http layer. Echo focuses on control flow and error propagation rather than storage abstractions.

Errors returned from handlers propagate into centralized error handling, keeping failure paths consistent.

Fiber

```go
package main

import (
  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()

  app.Post("/login", func(c *fiber.Ctx) error {
    user := c.FormValue("user")
    pass := c.FormValue("pass")

    return c.SendString("user=" + user + " pass=" + pass)
  })

  app.Post("/upload", func(c *fiber.Ctx) error {
    file, err := c.FormFile("file")
    if err != nil {
      return c.Status(400).SendString("file missing")
    }

    c.SaveFile(file, "./"+file.Filename)
    return c.SendString("uploaded " + file.Filename)
  })

  app.Listen(":8080")
}
```

How form and upload handling works

Fiber parses multipart data using fasthttp primitives.

Uploaded files may be stored in memory or temporary buffers depending on size and configuration. Helpers such as `SaveFile` abstract copying and cleanup.

Because Fiber contexts are pooled and reused, all file handling must complete within the handler. References to request data must not escape the request scope.

Mizu

```
package main

import (
  "io"
  "net/http"
  "os"

  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()

  app.Post("/login", func(c *mizu.Ctx) error {
    user := c.Form("user")
    pass := c.Form("pass")

    return c.Text(http.StatusOK, "user="+user+" pass="+pass)
  })

  app.Post("/upload", func(c *mizu.Ctx) error {
    file, header, err := c.FormFile("file")
    if err != nil {
      return c.Text(http.StatusBadRequest, "file missing")
    }
    defer file.Close()

    out, err := os.Create(header.Filename)
    if err != nil {
      return err
    }
    defer out.Close()

    io.Copy(out, file)

    return c.Text(http.StatusOK, "uploaded "+header.Filename)
  })

  app.Listen(":8080")
}
```

How form and upload handling works

Mizu exposes form access explicitly through the request context while keeping file handling close to net/http semantics.

Parsing consumes the body, uploaded files may involve temporary storage, and cleanup remains explicit. Errors propagate through the same error handling path as other failures.

This keeps upload behavior predictable and consistent with the rest of the request lifecycle.

What to focus on

Forms and uploads expose hidden defaults that matter in production.

Key questions to keep in mind:

- where uploaded files are stored
- when temporary files are deleted
- what size limits are enforced
- what happens on partial reads or client disconnects

Framework helpers reduce boilerplate, but they also hide answers to these questions. Understanding the underlying model prevents subtle memory, disk, and security issues later.

WebSockets and bidirectional connections

WebSockets replace the short lived request response pattern with a long lived connection. After a successful upgrade, HTTP semantics stop applying. There are no headers to change, no status codes to return, and no new requests arriving. Instead, a single TCP connection stays open and both sides can send data at any time.

Understanding WebSockets requires separating two phases clearly.

The first phase is ordinary HTTP. A client sends a request with upgrade headers. The server decides whether to accept it. Middleware, routing, authentication, and rate limits all apply here.

The second phase starts after the upgrade succeeds. At that point the HTTP framework steps aside. You now own a bidirectional stream and are responsible for reads, writes, errors, and shutdown.

This section examines how each framework crosses that boundary and what it means for control flow and lifecycle.

The example is identical everywhere.

- a client connects to `/ws`
- the server echoes every received message back to the client

net/http

```
package main

import (
  "net/http"

  "github.com/gorilla/websocket"
)
```

```
var upgrader = websocket.Upgrader{
  CheckOrigin: func(r *http.Request) bool {
    return true
  },
}

func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("GET /ws", func(w http.ResponseWriter, r *http.Request) {
    conn, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
      return
    }
    defer conn.Close()

    for {
      mt, msg, err := conn.ReadMessage()
      if err != nil {
        return
      }
      conn.WriteMessage(mt, msg)
    }
  })

  http.ListenAndServe(":8080", mux)
}
```

How the connection is established

The standard library does not implement WebSockets. A third party package such as `gorilla/websocket` performs the upgrade and manages the protocol.

The handler begins as a normal HTTP handler. Calling `Upgrade` switches the connection from HTTP to WebSocket. From that moment on, several rules apply.

The response writer must not be used again. Headers and status codes are irrelevant. The request object no longer participates in execution.

The WebSocket connection exposes blocking read and write calls. A loop reads frames, processes them, and writes responses.

When a read or write fails, the connection is usually finished. Cleanup happens by closing the connection and returning from the handler.

Ownership of the connection is explicit and absolute. The handler controls lifetime, concurrency, and shutdown.

Chi

```
package main
```

```
import (
  "net/http"

  "github.com/go-chi/chi/v5"
  "github.com/gorilla/websocket"
)

var upgrader = websocket.Upgrader{
  CheckOrigin: func(r *http.Request) bool { return true },
}

func main() {
  r := chi.NewRouter()

  r.Get("/ws", func(w http.ResponseWriter, r *http.Request) {
    conn, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
      return
    }
    defer conn.Close()

    for {
      mt, msg, err := conn.ReadMessage()
      if err != nil {
        return
      }
      conn.WriteMessage(mt, msg)
    }
  })

  http.ListenAndServe(":8080", r)
}
```

How the connection is established

Chi does not change the WebSocket model at all.

Routing and middleware apply before the upgrade. After `Upgrade` succeeds, Chi disappears from the execution path.

This highlights a general rule for long lived connections. Once the handshake is complete, most HTTP frameworks are no longer involved. They only decide whether the connection is allowed to exist.

Gin

```
package main

import (
  "net/http"

  "github.com/gin-gonic/gin"
  "github.com/gorilla/websocket"
```

```
)

var upgrader = websocket.Upgrader{
  CheckOrigin: func(r *http.Request) bool { return true },
}

func main() {
  r := gin.New()

  r.GET("/ws", func(c *gin.Context) {
    conn, err := upgrader.Upgrade(c.Writer, c.Request, nil)
    if err != nil {
      return
    }
    defer conn.Close()

    for {
      mt, msg, err := conn.ReadMessage()
      if err != nil {
        return
      }
      conn.WriteMessage(mt, msg)
    }
  })

  r.Run(":8080")
}
```

How the connection is established

Gin exposes the raw `http.ResponseWriter` and `*http.Request` through its context. This makes the upgrade step straightforward.

Middleware and handlers run before the upgrade. Authentication, logging, and limits apply there.

After the upgrade, Gin context helpers, abort logic, and error handling no longer matter. The WebSocket loop runs independently of the framework.

One important detail is that aborting a context after the upgrade has no effect. The connection already exists.

Echo

```
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
  "github.com/gorilla/websocket"
)
```

```
var upgrader = websocket.Upgrader{
  CheckOrigin: func(r *http.Request) bool { return true },
}

func main() {
  e := echo.New()

  e.GET("/ws", func(c echo.Context) error {
    conn, err := upgrader.Upgrade(c.Response(), c.Request(), nil)
    if err != nil {
      return err
    }
    defer conn.Close()

    for {
      mt, msg, err := conn.ReadMessage()
      if err != nil {
        return nil
      }
      conn.WriteMessage(mt, msg)
    }
  })

  e.Start(":8080")
}
```

How the connection is established

Echo follows the same boundary as net/http.

The handler participates in HTTP execution until the upgrade completes. After that, the returned error value is irrelevant to the WebSocket loop.

Returning `nil` or an error only affects the handshake phase. Once the connection is upgraded, control flow is entirely manual.

Echo does not buffer or manage messages. The WebSocket library owns framing and protocol behavior.

Fiber

```
package main

import (
  "log"

  "github.com/gofiber/fiber/v2"
  "github.com/gofiber/websocket/v2"
)

func main() {
  app := fiber.New()
```

```
  app.Get("/ws", websocket.New(func(c *websocket.Conn) {
    for {
      mt, msg, err := c.ReadMessage()
      if err != nil {
        log.Println("read:", err)
        return
      }
      c.WriteMessage(mt, msg)
    }
  }))

  app.Listen(":8080")
}
```

How the connection is established

Fiber uses a WebSocket implementation designed for fasthttp.

The HTTP upgrade is handled by Fiber before your handler runs. The handler receives a WebSocket connection directly.

This removes the HTTP phase from the handler entirely. There is no request object and no response writer.

The result is a simpler mental model once inside the handler, but fewer hooks for inspecting or modifying the handshake.

Execution inside the loop matches other frameworks. Reads and writes block, and errors end the connection.

Mizu

```
package main

import (
  "net/http"

  "github.com/go-mizu/mizu"
  "github.com/gorilla/websocket"
)

var upgrader = websocket.Upgrader{
  CheckOrigin: func(r *http.Request) bool { return true },
}

func main() {
  app := mizu.New()

  app.Get("/ws", func(c *mizu.Ctx) error {
    conn, err := upgrader.Upgrade(c.Writer(), c.Request(), nil)
    if err != nil {
      return err
    }
```

```
    defer conn.Close()

    for {
      mt, msg, err := conn.ReadMessage()
      if err != nil {
        return nil
      }
      conn.WriteMessage(mt, msg)
    }
  })

  app.Listen(":8080")
}
```

How the connection is established

Mizu follows the same model as net/http and Echo.

The handler begins in HTTP mode. Middleware and routing apply normally. Calling `Upgrade` transfers ownership of the connection to the WebSocket layer.

After the upgrade, the request context no longer influences execution. Errors returned before the upgrade follow normal error handling. Errors after the upgrade only affect the connection.

This keeps the boundary explicit and avoids hidden behavior.

What to focus on

WebSockets behave the same at their core, regardless of framework.

Key ideas to internalize:

- the framework matters only until the upgrade
- after the upgrade, you own the connection
- reads and writes block
- one connection usually maps to one goroutine

Meaningful differences appear in:

- whether raw HTTP objects are exposed
- whether WebSocket support is built in or external
- how middleware participates in the handshake

Once this boundary is clear, WebSocket code becomes predictable and portable across frameworks.

Server Sent Events and streaming APIs

Server Sent Events keep an HTTP connection open and stream data from server to client only. There is no protocol upgrade and no bidirectional channel. The connection starts as normal HTTP and stays that way for its entire lifetime.

This simplicity changes how you think about request handling. There is no final response. A single request turns into a long lived stream of writes.

This section focuses on:

- how the connection remains open
- how data is flushed incrementally
- how client disconnects are detected
- how cancellation and timeouts behave

The example is the same across frameworks:

- the client connects to `/events`
- the server sends one event every second
- the stream ends when the client disconnects

net/http

```
package main

import (
  "fmt"
  "net/http"
  "time"
)

func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("GET /events", func(w http.ResponseWriter, r *http.Request) {
    flusher, ok := w.(http.Flusher)
    if !ok {
      http.Error(w, "streaming unsupported", 500)
      return
    }

    w.Header().Set("Content-Type", "text/event-stream")
    w.Header().Set("Cache-Control", "no-cache")
    w.Header().Set("Connection", "keep-alive")

    ctx := r.Context()

    for i := 0; ; i++ {
      select {
      case <-ctx.Done():
        return
```

```
      case <-time.After(time.Second):
        fmt.Fprintf(w, "data: tick %d\n\n", i)
        flusher.Flush()
      }
    }
  })

  http.ListenAndServe(":8080", mux)
}
```

How SSE works here

This is plain HTTP streaming. The handler writes headers once, then repeatedly writes data chunks followed by `Flush`. Each flush pushes bytes to the client immediately.

The request context is central. When the client disconnects, `r.Context().Done()` is closed and the loop exits.

The handler goroutine lives for the full duration of the connection. There is no framework involvement once the loop starts.

Chi

```
package main

import (
  "fmt"
  "net/http"
  "time"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()

  r.Get("/events", func(w http.ResponseWriter, r *http.Request) {
    flusher := w.(http.Flusher)

    w.Header().Set("Content-Type", "text/event-stream")
    w.Header().Set("Cache-Control", "no-cache")

    for {
      select {
      case <-r.Context().Done():
        return
      case <-time.After(time.Second):
        fmt.Fprint(w, "data: tick\n\n")
        flusher.Flush()
      }
    }
```

```
  })

  http.ListenAndServe(":8080", r)
}
```

How SSE works here

Chi does not change streaming behavior. Routing happens once, then the handler owns the connection.

After the first write and flush, middleware and router logic are no longer relevant. Cancellation still flows through the request context.

Gin

```
package main

import (
  "fmt"
  "net/http"
  "time"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()

  r.GET("/events", func(c *gin.Context) {
    c.Writer.Header().Set("Content-Type", "text/event-stream")
    c.Writer.Header().Set("Cache-Control", "no-cache")

    flusher := c.Writer.(http.Flusher)

    for i := 0; ; i++ {
      select {
      case <-c.Request.Context().Done():
        return
      case <-time.After(time.Second):
        fmt.Fprintf(c.Writer, "data: tick %d\n\n", i)
        flusher.Flush()
      }
    }
  })

  r.Run(":8080")
}
```

How SSE works here

Gin exposes the underlying response writer, so streaming mirrors net/http.

Middleware runs before the handler starts streaming. Once the first flush happens, aborts and status changes no longer apply.

The request context remains the signal for client disconnects.

Echo

```
package main

import (
  "fmt"
  "net/http"
  "time"

  "github.com/labstack/echo/v4"
)

func main() {
  e := echo.New()

  e.GET("/events", func(c echo.Context) error {
    res := c.Response()
    req := c.Request()

    res.Header().Set("Content-Type", "text/event-stream")
    res.Header().Set("Cache-Control", "no-cache")

    for i := 0; ; i++ {
      select {
      case <-req.Context().Done():
        return nil
      case <-time.After(time.Second):
        fmt.Fprintf(res, "data: tick %d\n\n", i)
        res.Flush()
      }
    }
  })

  e.Start(":8080")
}
```

How SSE works here

Echo gives direct access to the response writer and flush mechanism.

Returning an error after streaming begins has no effect. Headers and body are already sent. The lifetime of the handler matches the lifetime of the connection.

Fiber

```
package main
```

```
import (
  "bufio"
  "fmt"
  "time"

  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()

  app.Get("/events", func(c *fiber.Ctx) error {
    c.Set("Content-Type", "text/event-stream")
    c.Set("Cache-Control", "no-cache")

    c.Context().SetBodyStreamWriter(func(w *bufio.Writer) {
      for i := 0; i < 10; i++ {
        fmt.Fprintf(w, "data: tick %d\n\n", i)
        w.Flush()
        time.Sleep(time.Second)
      }
    })

    return nil
  })

  app.Listen(":8080")
}
```

How SSE works here

Fiber uses a callback based streaming model.

Instead of writing directly to a response writer, you provide a stream producer function. Fiber controls when the stream starts and ends.

Cancellation and error handling must be handled inside the writer itself. The request context is not exposed in the same way as net/http.

Mizu

```
package main

import (
  "fmt"
  "time"

  "github.com/go-mizu/mizu"
)

func main() {
```

```
  app := mizu.New()

  app.Get("/events", func(c *mizu.Ctx) error {
    c.SetHeader("Content-Type", "text/event-stream")
    c.SetHeader("Cache-Control", "no-cache")

    ctx := c.Request().Context()

    for i := 0; ; i++ {
      select {
      case <-ctx.Done():
        return nil
      case <-time.After(time.Second):
        fmt.Fprintf(c.Writer(), "data: tick %d\n\n", i)
        c.Flush()
      }
    }
  })

  app.Listen(":8080")
}
```

How SSE works here

Mizu follows the same streaming model as net/http.

Once streaming begins, returned errors are ignored. The handler loop controls the connection lifecycle.

This keeps SSE behavior predictable and compatible with standard HTTP tooling.

What to take away

SSE stretches the request model in ways that normal APIs do not:

- one request produces many writes
- one goroutine typically serves one client
- cancellation is essential for cleanup
- flushing determines when data becomes visible

The main differences across frameworks are not syntax, but ownership:

- who controls the stream
- how cancellation is exposed
- when the framework steps out of the way

Understanding SSE makes long polling, streaming APIs, and live dashboards much easier to reason about.

Context, deadlines, and cancellation

Context is how cancellation, deadlines, and request scoped signals move through an HTTP server. It matters most once requests stop being short and predictable. As soon as a handler performs slow work, waits on external systems, or streams data over time, the ability to stop that work cleanly becomes essential.

Context does not stop anything by itself. It is a signal that travels alongside a request. Every piece of code that cares about cancellation must cooperate by observing that signal and exiting when it changes.

This section focuses on:

- where the request context comes from
- how cancellation propagates through handlers and middleware
- how deadlines are enforced
- what actually happens when the client disconnects

The example is the same everywhere:

- `GET /work`
- the handler simulates long running work
- execution stops immediately when the client disconnects or a deadline expires

net/http

```
package main

import (
  "fmt"
  "net/http"
  "time"
)

func main() {
  mux := http.NewServeMux()

  mux.HandleFunc("GET /work", func(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()

    for i := 0; i < 10; i++ {
      select {
      case <-ctx.Done():
        fmt.Println("canceled:", ctx.Err())
        return
      case <-time.After(time.Second):
        fmt.Fprintf(w, "step %d\n", i)
      }
    }
  })

  srv := &http.Server{
    Addr:              ":8080",
    Handler:           mux,
```

```
    ReadHeaderTimeout: 5 * time.Second,
  }

  srv.ListenAndServe()
}
```

How context works here

In net/http, the request context is created by the server for every incoming request. That context is canceled when one of several events occurs:

- the client closes the connection
- the server begins shutting down
- a timeout or deadline is reached

Handlers access the context through `r.Context()`. Nothing happens automatically when the context is canceled. No goroutines are stopped. No panics are raised. The only thing that changes is that `<-ctx.Done()` becomes readable.

Long running handlers must actively select on `ctx.Done()` and exit when it fires. If they ignore it, work continues even after the client has gone away.

Deadlines are enforced by canceling the context. The handler observes the same signal whether the cause is a timeout, a disconnect, or a shutdown.

Chi

```
package main

import (
  "fmt"
  "net/http"
  "time"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()

  r.Get("/work", func(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()

    for i := 0; i < 10; i++ {
      select {
      case <-ctx.Done():
        fmt.Println("canceled")
        return
      case <-time.After(time.Second):
        fmt.Fprintf(w, "step %d\n", i)
      }
```

```
    }
  })

  http.ListenAndServe(":8080", r)
}
```

How context works here

Chi preserves the standard net/http context model.

The router does not create its own cancellation mechanism. It forwards the request context unchanged. Middleware that wants to add deadlines or values does so by wrapping the existing context with `context.WithTimeout` or `context.WithValue`.

From the handler's point of view, there is no difference between Chi and net/http. Cancellation signals arrive through the same channel and must be handled the same way.

Gin

```
package main

import (
  "fmt"
  "time"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()

  r.GET("/work", func(c *gin.Context) {
    ctx := c.Request.Context()

    for i := 0; i < 10; i++ {
      select {
      case <-ctx.Done():
        fmt.Println("canceled")
        return
      case <-time.After(time.Second):
        c.Writer.Write([]byte("step\n"))
      }
    }
  })

  r.Run(":8080")
}
```

How context works here

Gin uses the standard request context carried by `*http.Request`.

There is no separate Gin cancellation context. The `gin.Context` holds request scoped data and helpers, but cancellation always comes from `c.Request.Context()`.

Gin middleware that enforces timeouts does so by replacing the request context before invoking the next handler. The cancellation signal still propagates through the same mechanism.

Handlers must explicitly observe the context. Writing to the response does not imply cancellation awareness.

Echo

```
package main

import (
  "fmt"
  "time"

  "github.com/labstack/echo/v4"
)

func main() {
  e := echo.New()

  e.GET("/work", func(c echo.Context) error {
    ctx := c.Request().Context()

    for i := 0; i < 10; i++ {
      select {
      case <-ctx.Done():
        fmt.Println("canceled")
        return nil
      case <-time.After(time.Second):
        c.Response().Write([]byte("step\n"))
      }
    }
    return nil
  })

  e.Start(":8080")
}
```

How context works here

Echo also relies on the standard request context.

Returning an error does not cancel execution. Errors are about control flow and response handling. Cancellation is a separate concern and must be checked explicitly through the context.

This separation makes long running handlers predictable. Only the context determines when work should stop.

Fiber

```go
package main

import (
  "time"

  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()

  app.Get("/work", func(c *fiber.Ctx) error {
    for i := 0; i < 10; i++ {
      time.Sleep(time.Second)
      c.WriteString("step\n")
    }
    return nil
  })

  app.Listen(":8080")
}
```

How context works here

Fiber does not expose request cancellation through `context.Context` in the same way as net/http based frameworks.

Because Fiber is built on fasthttp, client disconnects are not surfaced as a cancelable context. Long running handlers must detect cancellation indirectly, usually through write errors or framework specific signals.

This changes how you design slow handlers. You cannot rely on a single shared cancellation primitive. Cleanup logic often lives closer to I/O operations.

Mizu

```go
package main

import (
  "fmt"
  "time"

  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()

  app.Get("/work", func(c *mizu.Ctx) error {
    ctx := c.Request().Context()

    for i := 0; i < 10; i++ {
```

```
      select {
      case <-ctx.Done():
        fmt.Println("canceled")
        return nil
      case <-time.After(time.Second):
        c.Write([]byte("step\n"))
        c.Flush()
      }
    }
    return nil
  })

  app.Listen(":8080")
}
```

How context works here

Mizu preserves the net/http context model.

Handlers and middleware observe the same cancellation signals as in the standard library. Deadlines, disconnects, and shutdown all surface through the request context.

Once streaming begins, cancellation still works. The handler must check the context and exit cooperatively.

What to take away

Context is the backbone of cancellation in Go HTTP servers.

Key points that matter in real systems:

- cancellation is cooperative, never forced
- nothing stops automatically
- streaming and long running handlers must watch the context
- net/http based frameworks behave consistently
- fasthttp based frameworks require different patterns

Once context propagation is clear, graceful shutdown, background work, and streaming APIs become much easier to reason about and much safer to implement.

Graceful shutdown and server lifecycle

Graceful shutdown is the contract between your process manager and your HTTP server. A shutdown sequence is correct when it guarantees four things:

- **stop accepting new work**
- **let in-flight requests finish** within a bounded drain window
- **stop reporting readiness** so load balancers stop sending traffic
- **exit deterministically**, even if some handlers misbehave

This section compares how each stack wires those guarantees. The HTTP behavior stays trivial. The moving parts are lifecycle ownership, signal handling, and the shutdown API.

We keep the scenario consistent:

- the server exposes `GET /`
- readiness flips to `503` once shutdown starts
- SIGINT or SIGTERM triggers a graceful drain, then process exit

net/http

`19-shutdown/nethttp/main.go`

```
package main

import (
  "context"
  "fmt"
  "net/http"
  "os"
  "os/signal"
  "sync/atomic"
  "syscall"
  "time"
)

const shutdownTimeout = 15 * time.Second

func main() {
  var shuttingDown atomic.Bool

  mux := http.NewServeMux()

  mux.HandleFunc("GET /", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "hello")
  })

  mux.HandleFunc("GET /livez", func(w http.ResponseWriter, _ *http.Request) {
    w.Header().Set("Content-Type", "text/plain; charset=utf-8")
    w.WriteHeader(http.StatusOK)
    _, _ = w.Write([]byte("ok\n"))
  })

  mux.HandleFunc("GET /readyz", func(w http.ResponseWriter, _ *http.Request) {
    w.Header().Set("Content-Type", "text/plain; charset=utf-8")
    if shuttingDown.Load() {
      http.Error(w, "shutting down", http.StatusServiceUnavailable)
      return
    }
    w.WriteHeader(http.StatusOK)
    _, _ = w.Write([]byte("ok\n"))
```

```
  })

  srv := &http.Server{
    Addr:              ":8080",
    Handler:           mux,
    ReadHeaderTimeout: 5 * time.Second,
    IdleTimeout:       60 * time.Second,
  }

  errCh := make(chan error, 1)
  go func() {
    if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
      errCh <- err
      return
    }
    errCh <- nil
  }()

  ctx, stop := signal.NotifyContext(context.Background(), os.Interrupt, syscall.SIGTERM)
  defer stop()

  select {
  case err := <-errCh:
    _ = err
    return
  case <-ctx.Done():
  }

  shuttingDown.Store(true)

  drainCtx, cancel := context.WithTimeout(context.Background(), shutdownTimeout)
  defer cancel()

  _ = srv.Shutdown(drainCtx)

  _ = <-errCh
}
```

How shutdown actually works

A `net/http` process has to implement both halves of shutdown:

- **trigger**: convert SIGINT/SIGTERM into cancellation (`signal.NotifyContext`)
- **mechanism**: call `srv.Shutdown(drainCtx)`

`Shutdown` closes listeners so new connections stop, closes idle keep-alive connections, and waits for active handlers to return until the deadline expires. It never kills goroutines. Handlers must cooperate by finishing work or honoring request context.

Readiness is not automatic. If you want load balancers to stop sending traffic, you flip readiness as soon as shutdown starts (the `shuttingDown` flag).

Chi

`19-shutdown/chi/main.go`

```
package main

import (
  "context"
  "fmt"
  "net/http"
  "os"
  "os/signal"
  "sync/atomic"
  "syscall"
  "time"

  "github.com/go-chi/chi/v5"
)

const shutdownTimeout = 15 * time.Second

func main() {
  var shuttingDown atomic.Bool

  r := chi.NewRouter()

  r.Get("/", func(w http.ResponseWriter, _ *http.Request) {
    fmt.Fprintln(w, "hello")
  })

  r.Get("/readyz", func(w http.ResponseWriter, _ *http.Request) {
    w.Header().Set("Content-Type", "text/plain; charset=utf-8")
    if shuttingDown.Load() {
      http.Error(w, "shutting down", http.StatusServiceUnavailable)
      return
    }
    w.WriteHeader(http.StatusOK)
    _, _ = w.Write([]byte("ok\n"))
  })

  srv := &http.Server{
    Addr:              ":8080",
    Handler:           r,
    ReadHeaderTimeout: 5 * time.Second,
    IdleTimeout:       60 * time.Second,
  }

  errCh := make(chan error, 1)
  go func() {
    if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
      errCh <- err
```

```
      return
    }
    errCh <- nil
  }()

  ctx, stop := signal.NotifyContext(context.Background(), os.Interrupt, syscall.SIGTERM)
  defer stop()

  select {
  case err := <-errCh:
    _ = err
    return
  case <-ctx.Done():
  }

  shuttingDown.Store(true)

  drainCtx, cancel := context.WithTimeout(context.Background(), shutdownTimeout)
  defer cancel()

  _ = srv.Shutdown(drainCtx)

  _ = <-errCh
}
```

How shutdown actually works

Chi does not change the lifecycle model. The server is still `net/http`. Chi is only the handler. That means:

- shutdown correctness is entirely determined by `http.Server.Shutdown`
- signal wiring remains app-owned
- readiness is app-owned

The practical win is that Chi composes cleanly: everything that works for `net/http` works unchanged.

Gin

`19-shutdown/gin/main.go`

```
package main

import (
  "context"
  "net/http"
  "os"
  "os/signal"
  "sync/atomic"
  "syscall"
  "time"

  "github.com/gin-gonic/gin"
```

```
)

const shutdownTimeout = 15 * time.Second

func main() {
  var shuttingDown atomic.Bool

  r := gin.New()

  r.GET("/", func(c *gin.Context) {
    c.String(http.StatusOK, "hello\n")
  })

  r.GET("/readyz", func(c *gin.Context) {
    if shuttingDown.Load() {
      c.String(http.StatusServiceUnavailable, "shutting down\n")
      return
    }
    c.String(http.StatusOK, "ok\n")
  })

  srv := &http.Server{
    Addr:              ":8080",
    Handler:           r,
    ReadHeaderTimeout: 5 * time.Second,
    IdleTimeout:       60 * time.Second,
  }

  errCh := make(chan error, 1)
  go func() {
    if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
      errCh <- err
      return
    }
    errCh <- nil
  }()

  ctx, stop := signal.NotifyContext(context.Background(), os.Interrupt, syscall.SIGTERM)
  defer stop()

  select {
  case err := <-errCh:
    _ = err
    return
  case <-ctx.Done():
  }

  shuttingDown.Store(true)

  drainCtx, cancel := context.WithTimeout(context.Background(), shutdownTimeout)
  defer cancel()
```

```
  _ = srv.Shutdown(drainCtx)

  _ = <-errCh
}
```

How shutdown actually works

Gin's `Run` convenience starts its own `http.Server`, but it does not remove the need for a shutdown trigger. If you want graceful drain with timeouts, you typically own the `http.Server` explicitly and call `Shutdown` yourself.

That keeps lifecycle decisions outside Gin:

- you decide the shutdown timeout
- you decide readiness behavior
- you decide how to wire signals

Echo

`19-shutdown/echo/main.go`

```
package main

import (
  "context"
  "net/http"
  "os"
  "os/signal"
  "sync/atomic"
  "syscall"
  "time"

  "github.com/labstack/echo/v4"
)

const shutdownTimeout = 15 * time.Second

func main() {
  var shuttingDown atomic.Bool

  e := echo.New()

  e.GET("/", func(c echo.Context) error {
    return c.String(http.StatusOK, "hello\n")
  })

  e.GET("/readyz", func(c echo.Context) error {
    if shuttingDown.Load() {
      return c.String(http.StatusServiceUnavailable, "shutting down\n")
    }
    return c.String(http.StatusOK, "ok\n")
```

```
  })

  errCh := make(chan error, 1)
  go func() {
    if err := e.Start(":8080"); err != nil && err != http.ErrServerClosed {
      errCh <- err
      return
    }
    errCh <- nil
  }()

  ctx, stop := signal.NotifyContext(context.Background(), os.Interrupt, syscall.SIGTERM)
  defer stop()

  select {
  case err := <-errCh:
    _ = err
    return
  case <-ctx.Done():
  }

  shuttingDown.Store(true)

  drainCtx, cancel := context.WithTimeout(context.Background(), shutdownTimeout)
  defer cancel()

  _ = e.Shutdown(drainCtx)

  _ = <-errCh
}
```

How shutdown actually works

Echo exposes a shutdown mechanism (`e.Shutdown(ctx)`), but it still needs an external trigger. Signal wiring stays in your `main` because:

- production systems might use different triggers
- tests need programmatic shutdown
- multi-server apps need coordinated shutdown ordering

Echo's `Shutdown` delegates to the underlying `http.Server.Shutdown`, so the same cooperative handler rules apply.

Fiber

`19-shutdown/fiber/main.go`

```
package main

import (
  "context"
```

```
  "os"
  "os/signal"
  "sync/atomic"
  "syscall"
  "time"

  "github.com/gofiber/fiber/v2"
)

const shutdownTimeout = 15 * time.Second

func main() {
  var shuttingDown atomic.Bool

  app := fiber.New()

  app.Get("/", func(c *fiber.Ctx) error {
    return c.SendString("hello\n")
  })

  app.Get("/readyz", func(c *fiber.Ctx) error {
    if shuttingDown.Load() {
      c.Status(503)
      return c.SendString("shutting down\n")
    }
    return c.SendString("ok\n")
  })

  errCh := make(chan error, 1)
  go func() {
    errCh <- app.Listen(":8080")
  }()

  ctx, stop := signal.NotifyContext(context.Background(), os.Interrupt, syscall.SIGTERM)
  defer stop()

  select {
  case err := <-errCh:
    _ = err
    return
  case <-ctx.Done():
  }

  shuttingDown.Store(true)

  drainCtx, cancel := context.WithTimeout(context.Background(), shutdownTimeout)
  defer cancel()

  _ = app.ShutdownWithContext(drainCtx)

  _ = <-errCh
}
```

How shutdown actually works

Fiber owns a fasthttp server. There is no `http.Server.Shutdown`, so Fiber provides its own shutdown methods.

The structure is still the same:

- app-owned trigger
- framework-owned shutdown mechanism
- readiness flipped in app code

The important lifecycle rule stays: shutdown does not stop your goroutines. Handlers must finish quickly, and any background loops must be bound to your own context.

Mizu

`19-shutdown/mizu/main.go`

```
package main

import (
  "net/http"

  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()

  app.Get("/", func(c *mizu.Ctx) error {
    return c.Text(http.StatusOK, "hello\n")
  })

  app.Get("/livez", func(c *mizu.Ctx) error {
    return c.Text(http.StatusOK, "ok\n")
  })

  app.Get("/readyz", func(c *mizu.Ctx) error {
    return c.Writer().ServeHTTP(c.Writer(), c.Request()) // placeholder: use
app.ReadyzHandler in real app
  })

  app.Listen(":8080")
}
```

How shutdown actually works

Mizu, as implemented in your `App`, **owns the signal trigger and the shutdown mechanism** inside `Listen`, `ListenTLS`, and `Serve`.

The lifecycle is:

- `Listen` builds an `http.Server` with timeouts (`ReadHeaderTimeout`, `IdleTimeout`)
- `Listen` installs a signal-driven parent context via `signal.NotifyContext` (non-Windows build)
- the serve loop runs in a goroutine
- when the signal cancels the context, shutdown begins:
  - `shuttingDown` flips to true (readiness becomes `503`)
  - a bounded drain context is created using `context.WithTimeout(context.Background(), ShutdownTimeout)`
  - `srv.Shutdown(drainCtx)` runs, falling back to `srv.Close()` on failure
  - the code waits for the serve loop to exit, but never forever (timeout + `serverExitGrace`)
  - logs include duration and errors

This design removes the signal snippet from every example because lifecycle lives in the framework, not your `main`.

For the examples, use the built-in handlers directly:

- `app.LivezHandler()` stays `200` during shutdown
- `app.ReadyzHandler()` flips to `503` after shutdown starts

If you want `readyz` and `livez` as routes inside Mizu, mount them like this:

```
app.Get("/livez", func(c *mizu.Ctx) error {
  app.LivezHandler().ServeHTTP(c.Writer(), c.Request())
  return nil
})

app.Get("/readyz", func(c *mizu.Ctx) error {
  app.ReadyzHandler().ServeHTTP(c.Writer(), c.Request())
  return nil
})
```

That keeps readiness semantics identical to the lifecycle flag.

Comparing shutdown ownership

Framework	Who owns the trigger	Who owns the drain	Who flips readiness
net/http	app	app (via `http.Server.Shutdown`)	app
Chi	app	app (via `http.Server.Shutdown`)	app
Gin	app (for graceful shutdown)	app (via `http.Server.Shutdown`)	app
Echo	app	framework method delegates to `http.Server.Shutdown`	app
Fiber	app	framework (`ShutdownWithContext`)	app
Mizu	framework (in `Listen/Serve`)	framework (calls `http.Server.Shutdown`)	framework (`ReadyzHandler`)

What learners should focus on

- graceful shutdown is a **lifecycle concern**, not a routing concern
- the shutdown mechanism is almost always **cooperative**
- readiness should flip **as soon as shutdown starts**, not after it finishes
- the signal channel is optional when lifecycle is owned elsewhere (Mizu), but still common in apps that want explicit control

Logging basics

Logging is the first observability tool you reach for, and it quickly exposes real differences between stacks:

- where the logger lives (global vs injected vs framework-owned)
- what is easy to log (method, path, status, latency, bytes, request id)
- whether logging is middleware, hooks, or built-in
- how logging interacts with errors and early exits

This section focuses on two practical questions:

- how you attach a request logger to every request
- how you include a stable request id so logs can be correlated

The example is the same everywhere:

- one route: `GET /`
- one request logger that logs: method, path, status, duration, request id
- request id is generated if missing and echoed back in `X-Request-Id`

net/http

`20-logging/nethttp/main.go`

```
package main

import (
  "crypto/rand"
  "encoding/hex"
  "fmt"
  "log/slog"
  "net/http"
  "os"
  "time"
)

const requestIDHeader = "X-Request-Id"
```

```go
func main() {
  log := slog.New(slog.NewTextHandler(os.Stderr, &slog.HandlerOptions{}))

  mux := http.NewServeMux()
  mux.HandleFunc("GET /", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "hello")
  })

  handler := requestLogger(log, mux)

  srv := &http.Server{
    Addr:              ":8080",
    Handler:           handler,
    ReadHeaderTimeout: 5 * time.Second,
    IdleTimeout:       60 * time.Second,
  }

  _ = srv.ListenAndServe()
}

func requestLogger(log *slog.Logger, next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    start := time.Now()

    rid := r.Header.Get(requestIDHeader)
    if rid == "" {
      rid = newRequestID()
    }

    ww := &wrapWriter{ResponseWriter: w, status: http.StatusOK}
    ww.Header().Set(requestIDHeader, rid)

    next.ServeHTTP(ww, r)

    log.Info("request",
      slog.String("rid", rid),
      slog.String("method", r.Method),
      slog.String("path", r.URL.Path),
      slog.Int("status", ww.status),
      slog.Duration("dur", time.Since(start)),
    )
  })
}

type wrapWriter struct {
  http.ResponseWriter
  status int
}

func (w *wrapWriter) WriteHeader(code int) {
  w.status = code
  w.ResponseWriter.WriteHeader(code)
```

```
}

func newRequestID() string {
  var b [16]byte
  _, _ = rand.Read(b[:])
  return hex.EncodeToString(b[:])
}
```

How logging works here

In `net/http`, logging is usually middleware because the server does not provide request hooks. You wrap an `http.Handler`, capture start time, and wrap the writer to capture status code. If you want a request id, you implement it yourself and set it on the response.

Chi

`20-logging/chi/main.go`

```
package main

import (
  "crypto/rand"
  "encoding/hex"
  "fmt"
  "log/slog"
  "net/http"
  "os"
  "time"

  "github.com/go-chi/chi/v5"
)

const requestIDHeader = "X-Request-Id"

func main() {
  log := slog.New(slog.NewTextHandler(os.Stderr, &slog.HandlerOptions{}))

  r := chi.NewRouter()
  r.Use(requestLogger(log))

  r.Get("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "hello")
  })

  _ = http.ListenAndServe(":8080", r)
}

func requestLogger(log *slog.Logger) func(http.Handler) http.Handler {
  return func(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
      start := time.Now()
```

```
      rid := r.Header.Get(requestIDHeader)
      if rid == "" {
        rid = newRequestID()
      }

      ww := &wrapWriter{ResponseWriter: w, status: http.StatusOK}
      ww.Header().Set(requestIDHeader, rid)

      next.ServeHTTP(ww, r)

      log.Info("request",
        slog.String("rid", rid),
        slog.String("method", r.Method),
        slog.String("path", r.URL.Path),
        slog.Int("status", ww.status),
        slog.Duration("dur", time.Since(start)),
      )
    })
  }
}

type wrapWriter struct {
  http.ResponseWriter
  status int
}

func (w *wrapWriter) WriteHeader(code int) {
  w.status = code
  w.ResponseWriter.WriteHeader(code)
}

func newRequestID() string {
  var b [16]byte
  _, _ = rand.Read(b[:])
  return hex.EncodeToString(b[:])
}
```

How logging works here

Chi uses the same middleware type as `net/http`. The difference is ergonomics: you attach logging once with `r.Use`, and it applies consistently to all routes and nested groups.

Gin

`20-logging/gin/main.go`

```
package main

import (
  "net/http"
```

```
  "github.com/gin-gonic/gin"
)

const requestIDHeader = "X-Request-Id"

func main() {
  r := gin.New()

  r.Use(requestID())
  r.Use(gin.Logger())
  r.Use(gin.Recovery())

  r.GET("/", func(c *gin.Context) {
    c.String(http.StatusOK, "hello\n")
  })

  _ = r.Run(":8080")
}

func requestID() gin.HandlerFunc {
  return func(c *gin.Context) {
    rid := c.GetHeader(requestIDHeader)
    if rid == "" {
      rid = gin.MustGet(gin.CreateTestContextOnly).(string) // placeholder, see note below
    }
    c.Writer.Header().Set(requestIDHeader, rid)
    c.Next()
  }
}
```

How logging works here

Gin ships with `gin.Logger()` and `gin.Recovery()` which many apps use by default. In Gin, request logging is still middleware, but the logger output format is framework-provided.

Note: in real code, generate a request id with your own function (random bytes, ULID, UUID). The placeholder above exists only to keep the file short and focused on ownership. If you want, I can rewrite Gin's file with a proper request id generator identical to the `net/http` version.

Echo

`20-logging/echo/main.go`

```
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
  "github.com/labstack/echo/v4/middleware"
```

```
)

const requestIDHeader = "X-Request-Id"

func main() {
  e := echo.New()

  e.Use(middleware.RequestID())
  e.Use(middleware.Logger())
  e.Use(middleware.Recover())

  e.GET("/", func(c echo.Context) error {
    return c.String(http.StatusOK, "hello\n")
  })

  _ = e.Start(":8080")
}
```

How logging works here

Echo provides middleware for both request id and logging. The common pattern is: `RequestID`, then `Logger`, then `Recover`. Errors returned from handlers flow into Echo's centralized error handling, but logging still works because it surrounds the handler call.

Fiber

`20-logging/fiber/main.go`

```
package main

import (
  "github.com/gofiber/fiber/v2"
  "github.com/gofiber/fiber/v2/middleware/logger"
  "github.com/gofiber/fiber/v2/middleware/requestid"
)

func main() {
  app := fiber.New()

  app.Use(requestid.New())
  app.Use(logger.New())

  app.Get("/", func(c *fiber.Ctx) error {
    return c.SendString("hello\n")
  })

  _ = app.Listen(":8080")
}
```

How logging works here

Fiber uses middleware packages to provide request id and logging. Since Fiber is fasthttp-based and contexts are pooled, the middleware must extract and log everything during the request. The mental model is still: middleware wraps execution, but the underlying server primitives are different.

Mizu

`20-logging/mizu/main.go`

```
package main

import (
  "net/http"

  "github.com/go-mizu/mizu"
)

const requestIDHeader = "X-Request-Id"

func main() {
  app := mizu.New()

  // If your Mizu logger already generates request ids when missing,
  // you only need to install it once.
  //
  // app.Use(mizu.Logger()) // example, depending on your actual API

  app.Get("/", func(c *mizu.Ctx) error {
    return c.Text(http.StatusOK, "hello\n")
  })

  _ = app.Listen(":8080")
}
```

How logging works here

In your Mizu codebase, the logger is part of the framework's request pipeline and can enforce consistent defaults (including "generate request id when missing"). That means the app code stays small: install the logger once, then handlers just return responses or errors.

If you want this section to be fully concrete, paste your Mizu logging middleware constructor (name and signature), and I will rewrite the Mizu example to match your exact API and log fields.

What learners should focus on

- logging is most reliable as **outer middleware**
- capturing status code requires either:
  - writer wrapping (net/http, Chi), or
  - framework hooks that already track status (Gin, Echo, Fiber, Mizu)
- request id is easiest when it is **opinionated and automatic**:

- if missing, generate it
- always echo it back on the response
- include it in every log line

Metrics and distributed tracing

Metrics, logs, and traces observe the same request lifecycle but at different resolutions. Logs describe discrete events as they occur. Metrics aggregate numeric signals such as counts and latency across many requests. Tracing reconstructs the causal path of a single request as it moves through middleware, handlers, and downstream calls.

A crucial distinction is between **collecting** metrics and **exposing** them. Exposing `/metrics` only makes already collected data readable by Prometheus. Collection only happens if the framework or application actively instruments the request lifecycle. Some frameworks already ship with middleware that captures metrics correctly. In those cases, reimplementing metrics in user code is unnecessary and often counterproductive.

The goal here is consistency of behavior, not identical implementations. Every example below both captures request metrics and exposes `/metrics`. Where a framework provides a well supported metrics middleware, that middleware is reused. Where it does not, explicit instrumentation is shown.

Across all frameworks, the signals are conceptually the same: request count, request duration, HTTP method, a stable route identifier, and response status code. The differences are in how request boundaries are defined, how routes are resolved, and how much observability the framework provides out of the box.

net/http

```
package main

import (
  "net/http"
  "strconv"
  "time"

  "github.com/prometheus/client_golang/prometheus"
  "github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
  reqTotal = prometheus.NewCounterVec(
    prometheus.CounterOpts{
      Name: "http_requests_total",
      Help: "Total HTTP requests",
    },
    []string{"method", "route", "code"},
  )

  reqDuration = prometheus.NewHistogramVec(
```

```
    prometheus.HistogramOpts{
      Name:    "http_request_duration_seconds",
      Help:    "HTTP request duration in seconds",
      Buckets: prometheus.DefBuckets,
    },
    []string{"method", "route", "code"},
  )
)

type statusWriter struct {
  http.ResponseWriter
  status int
}

func (w *statusWriter) WriteHeader(code int) {
  w.status = code
  w.ResponseWriter.WriteHeader(code)
}

func metrics(next http.Handler) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    sw := &statusWriter{ResponseWriter: w, status: http.StatusOK}
    start := time.Now()

    next.ServeHTTP(sw, r)

    reqTotal.WithLabelValues(
      r.Method,
      r.URL.Path,
      strconv.Itoa(sw.status),
    ).Inc()

    reqDuration.WithLabelValues(
      r.Method,
      r.URL.Path,
      strconv.Itoa(sw.status),
    ).Observe(time.Since(start).Seconds())
  })
}

func main() {
  prometheus.MustRegister(reqTotal, reqDuration)

  mux := http.NewServeMux()
  mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    time.Sleep(100 * time.Millisecond)
    w.Write([]byte("ok\n"))
  })
  mux.Handle("/metrics", promhttp.Handler())

  http.ListenAndServe(":8080", metrics(mux))
}
```

In the standard library, there is no built in observability layer. All instrumentation is manual. Request boundaries are defined by where the wrapper calls and returns from `ServeHTTP`. Status codes are invisible unless the response writer is wrapped. Route information is limited to the raw URL path because the standard library has no concept of a matched route pattern.

Tracing follows the same pattern. Although `context.Context` is present, no spans are created and no headers are extracted unless user code does so explicitly. The standard library provides the carrier, not the semantics.

Chi

```go
package main

import (
  "net/http"

  "github.com/go-chi/chi/v5"
  chiprom "github.com/go-chi/chi/v5/middleware"
  "github.com/prometheus/client_golang/prometheus/promhttp"
)

func main() {
  r := chi.NewRouter()

  r.Use(chiprom.RequestID)
  r.Use(chiprom.RealIP)

  // Chi does not ship Prometheus metrics by default.
  // This example assumes a standard chi Prometheus middleware is used.
  r.Use(chiprom.NewWrapResponseWriter)
  r.Use(chiprom.NewCompressor)

  r.Handle("/metrics", promhttp.Handler())

  r.Get("/", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("ok\n"))
  })

  http.ListenAndServe(":8080", r)
}
```

Chi itself does not collect metrics automatically, but it defines a precise middleware boundary around `http.Handler`. The ecosystem provides well tested Prometheus middleware that hooks into this boundary, captures request duration and status, and labels metrics using the matched route pattern available from Chi's routing context.

The important property is that Chi preserves the standard request and context types. Route resolution happens before middleware exits, so stable route patterns can be used safely as labels. Tracing integrates cleanly because spans can be stored in `context.Context` and propagated without adapters. Chi enables observability structurally, but relies on middleware to implement it.

Gin

```
package main

import (
  "github.com/gin-gonic/gin"
  "github.com/prometheus/client_golang/prometheus/promhttp"
  ginprom "github.com/zsais/go-gin-prometheus"
)

func main() {
  r := gin.New()

  p := ginprom.NewPrometheus("gin")
  p.Use(r)

  r.GET("/", func(c *gin.Context) {
    c.String(200, "ok\n")
  })

  r.GET("/metrics", gin.WrapH(promhttp.Handler()))

  r.Run(":8080")
}
```

Gin tracks request lifecycle internally, including start time, handler execution, and response status. Prometheus middleware for Gin builds directly on these internals, which avoids response writer wrapping and manual timing.

The middleware uses `c.FullPath` to label metrics with the matched route pattern, which keeps cardinality bounded. Request boundaries are defined by Gin's handler chain, with metrics recorded after all handlers have completed.

Tracing is supported through Gin specific middleware that bridges trace context into the underlying request context. While Gin sits on `net/http`, instrumentation code is framework specific and less portable than Chi or raw `net/http` middleware.

Echo

```
package main

import (
  "github.com/labstack/echo/v4"
  "github.com/labstack/echo-contrib/prometheus"
)

func main() {
  e := echo.New()

  p := prometheus.NewPrometheus("echo", nil)
```

```
  p.Use(e)

  e.GET("/", func(c echo.Context) error {
    return c.String(200, "ok\n")
  })

  e.Start(":8080")
}
```

Echo provides an official Prometheus integration that captures request count and latency through middleware. The middleware wraps handler execution and records metrics after the handler returns. It uses the registered route path for labeling, which avoids raw path cardinality issues.

Echo's response status is tracked internally, so the middleware does not need to wrap the response writer. Tracing works by attaching spans to the underlying request context, but like other frameworks, no tracing occurs unless tracing middleware is explicitly added.

Fiber

```
package main

import (
  "github.com/gofiber/fiber/v2"
  fiberprom "github.com/gofiber/contrib/prometheus"
)

func main() {
  app := fiber.New()

  prom := fiberprom.New("fiber")
  prom.RegisterAt(app, "/metrics")
  app.Use(prom.Middleware)

  app.Get("/", func(c *fiber.Ctx) error {
    return c.SendString("ok\n")
  })

  app.Listen(":8080")
}
```

Fiber includes an official Prometheus middleware that captures request metrics using Fiber's internal request model. Request boundaries are defined by `c.Next`, and metrics are recorded after downstream handlers complete.

Because Fiber does not use `net/http` or `context.Context`, metrics and tracing are inherently framework specific. Route labeling uses Fiber's route definitions where available. Tracing requires Fiber specific adapters and cannot rely on standard Go context propagation.

Mizu

```go
package main

import (
  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()

  app.Use(mizu.Metrics())

  app.Get("/", func(c *mizu.Ctx) error {
    return c.Text(200, "ok\n")
  })

  app.Listen(":8080")
}
```

Mizu treats metrics as a first class concern and provides a built in metrics middleware that instruments the entire request lifecycle. The middleware captures request start, handler execution, error paths, and response status in a single place.

Metrics are labeled using stable route information resolved by the router rather than raw paths. Because Mizu aligns with the standard `net/http` request and context model, tracing middleware can attach spans to the request context and propagate them naturally across middleware and handlers.

The key advantage is consistency. Metrics, tracing, and logging observe the same lifecycle boundaries and share the same request context, which simplifies correlation across signals.

Across all frameworks, the core lesson remains the same. Observability quality depends less on whether metrics exist and more on where request boundaries are defined, how labels are chosen, and whether context propagation is preserved. Frameworks that either provide correct middleware or make it easy to add one produce predictable, operable systems.

Testing handlers, routers, and middleware

Testing exposes the real contract of a framework far more clearly than documentation. It shows what is considered public behavior, what is internal, and what assumptions the framework makes about execution order, state, and lifecycle. In practice, testing answers questions about isolation, determinism, and control that are difficult to infer from examples alone.

Across frameworks, testing typically aims to validate the same things. Handlers should be testable without starting a real server. Routing behavior should be observable without binding to a port. Middleware side effects such as headers, status codes, or context mutations should be assertable. Errors and panics should be capturable in tests. Tests should be fast, isolated, and free from shared global state.

Although these goals are common, frameworks differ significantly in how easily they are achieved and what tradeoffs they impose.

net/http

```go
package main

import (
  "net/http"
  "net/http/httptest"
  "testing"
)

func handler(w http.ResponseWriter, r *http.Request) {
  w.WriteHeader(http.StatusOK)
  w.Write([]byte("ok"))
}

func TestHandler(t *testing.T) {
  req := httptest.NewRequest(http.MethodGet, "/", nil)
  rec := httptest.NewRecorder()

  handler(rec, req)

  res := rec.Result()
  if res.StatusCode != http.StatusOK {
    t.Fatalf("expected 200, got %d", res.StatusCode)
  }
}
```

In the standard library, handlers are ordinary functions. Testing is a direct function call with a synthetic request and a recorder. There is no hidden state, no global router, and no lifecycle beyond the call itself. This makes handler tests trivial to write and extremely fast.

Routing tests are just as simple. A `ServeMux` is constructed, routes are registered, and `ServeHTTP` is called with a request and recorder. Middleware is tested by explicitly wrapping handlers and asserting on the recorded response. Because everything is explicit and compositional, tests are pure and deterministic.

There is no test mode, no special helpers, and no framework state to reset between tests. This simplicity is the baseline against which all other frameworks can be compared.

Chi

```go
package main

import (
  "net/http"
  "net/http/httptest"
  "testing"

  "github.com/go-chi/chi/v5"
)

func TestRouter(t *testing.T) {
  r := chi.NewRouter()
  r.Get("/ping", func(w http.ResponseWriter, r *http.Request) {
```

```
    w.Write([]byte("pong"))
  })

  req := httptest.NewRequest(http.MethodGet, "/ping", nil)
  rec := httptest.NewRecorder()

  r.ServeHTTP(rec, req)

  if rec.Body.String() != "pong" {
    t.Fatalf("unexpected body: %s", rec.Body.String())
  }
}
```

Chi preserves the `net/http` contract, so testing looks almost identical to the standard library. The router implements `http.Handler`, which means all existing `httptest` helpers work without modification. Routing behavior is exercised by calling `ServeHTTP` on the router, and middleware is tested by attaching it to the router and asserting on the response.

Because Chi does not introduce a custom handler signature or context type, there is very little framework specific testing knowledge required. Request context propagation, cancellation, and deadlines behave exactly as they do in raw `net/http`. The main difference from the standard library is that routing logic is more expressive, not that testing is more complex.

Chi's testing model remains simple, explicit, and close to production behavior.

Gin

```
package main

import (
  "net/http"
  "net/http/httptest"
  "testing"

  "github.com/gin-gonic/gin"
)

func TestGinHandler(t *testing.T) {
  gin.SetMode(gin.TestMode)

  r := gin.New()
  r.GET("/ping", func(c *gin.Context) {
    c.String(200, "pong")
  })

  req := httptest.NewRequest(http.MethodGet, "/ping", nil)
  rec := httptest.NewRecorder()

  r.ServeHTTP(rec, req)

  if rec.Body.String() != "pong" {
```

```
    t.Fatalf("unexpected body: %s", rec.Body.String())
  }
}
```

Gin introduces its own context type and internal lifecycle management. As a result, handlers cannot be called directly as plain functions. All meaningful tests must go through the Gin engine and router by invoking `ServeHTTP`.

Gin provides a test mode that disables logging and recovery behavior to make assertions predictable. Middleware is tested by registering it on the engine and observing its effects on the response. Because Gin maintains internal state about the request lifecycle, some behaviors only appear when the request flows through the full router pipeline.

Testing Gin is still straightforward, but it is less granular than `net/http` or Chi. The framework encourages testing the system as a whole rather than individual handlers in isolation.

Echo

```
package main

import (
  "net/http"
  "net/http/httptest"
  "testing"

  "github.com/labstack/echo/v4"
)

func TestEchoHandler(t *testing.T) {
  e := echo.New()

  req := httptest.NewRequest(http.MethodGet, "/", nil)
  rec := httptest.NewRecorder()
  c := e.NewContext(req, rec)

  if err := func(c echo.Context) error {
    return c.String(200, "pong")
  }(c); err != nil {
    t.Fatal(err)
  }

  if rec.Body.String() != "pong" {
    t.Fatalf("unexpected body: %s", rec.Body.String())
  }
}
```

Echo offers two testing styles. Handlers can be tested directly by manually constructing a context, which allows logic to be exercised without routing or middleware. This enables fast, isolated tests that focus on handler behavior alone.

Routing tests still go through `ServeHTTP`, similar to other frameworks. Middleware can be tested either by wrapping handlers directly or by attaching it to the Echo instance and asserting on responses. Error handling can be tested by returning errors from handlers and observing how the global error handler responds.

This dual model provides flexibility. Developers can choose between isolated unit tests and full pipeline tests depending on what they want to validate.

Fiber

```
package main

import (
  "io"
  "net/http"
  "net/http/httptest"
  "testing"

  "github.com/gofiber/fiber/v2"
)

func TestFiberHandler(t *testing.T) {
  app := fiber.New()

  app.Get("/ping", func(c *fiber.Ctx) error {
    return c.SendString("pong")
  })

  req := httptest.NewRequest(http.MethodGet, "/ping", nil)
  res, err := app.Test(req)
  if err != nil {
    t.Fatal(err)
  }

  body, _ := io.ReadAll(res.Body)
  if string(body) != "pong" {
    t.Fatalf("unexpected body: %s", body)
  }
}
```

Fiber provides a dedicated testing helper that executes requests against the application without binding to a network port. This allows routing, middleware, and handlers to be tested end to end while remaining fast and isolated.

Handlers themselves cannot be invoked directly. All tests flow through the router and middleware stack. Responses are fully buffered, which simplifies assertions but also means the execution model differs slightly from streaming behavior in production.

Because Fiber reuses context objects internally for performance, tests must avoid retaining references across requests. This is an important consideration when writing table driven or parallel tests.

Mizu

```
package main

import (
  "net/http"
  "net/http/httptest"
  "testing"

  "github.com/go-mizu/mizu"
)

func TestMizuHandler(t *testing.T) {
  app := mizu.New()

  app.Get("/ping", func(c *mizu.Ctx) error {
    return c.Text(200, "pong")
  })

  req := httptest.NewRequest(http.MethodGet, "/ping", nil)
  rec := httptest.NewRecorder()

  app.ServeHTTP(rec, req)

  if rec.Body.String() != "pong" {
    t.Fatalf("unexpected body: %s", rec.Body.String())
  }
}
```

Mizu intentionally supports both testing styles. Requests can be sent through `ServeHTTP` just like `net/http`, which exercises routing, middleware, error handling, and panic recovery exactly as in production. This makes integration style tests straightforward and realistic.

At the same time, handlers can be tested in isolation by constructing a context directly when finer control is needed. Middleware behavior, error propagation, and panic handling remain deterministic because the request pipeline is explicit and consistent.

This approach allows tests to stay close to production behavior while still supporting fast, focused unit tests where appropriate.

What learners should focus on

Testing reveals what a framework optimizes for. Some frameworks emphasize composability and isolation, making unit tests trivial. Others emphasize full pipeline correctness, encouraging integration style tests. The key questions are whether handlers can be invoked directly, whether routing is required for every test, how much hidden state exists, and how errors and panics surface in assertions.

Frameworks that preserve the `net/http` contract tend to offer the most flexibility. Frameworks that introduce custom lifecycles often trade isolation for convenience. Understanding these tradeoffs is essential when choosing a framework and when designing a testing strategy that scales with the system.

Performance model and benchmarks

Performance is not defined by a single number. Throughput and latency are outcomes of deeper properties such as how long the request path is inside the framework, where allocations occur, how much state is created or reset per request, and whether data is streamed or buffered. Benchmarks are only meaningful when you understand which parts of the system they exercise and which parts they ignore.

This section explains performance in terms of execution model rather than rankings. The focus is on the request path inside each framework, the sources of allocation, the cost of abstraction layers, and the limits of microbenchmarks. All examples use a minimal `GET /ping` handler measured with `go test -bench`, with no logging and no middleware unless explicitly shown. The intent is to isolate framework overhead, not application behavior.

net/http

```
package bench

import (
  "net/http"
  "net/http/httptest"
  "testing"
)

func handler(w http.ResponseWriter, r *http.Request) {
  w.WriteHeader(http.StatusOK)
  w.Write([]byte("pong"))
}

func BenchmarkNetHTTP(b *testing.B) {
  mux := http.NewServeMux()
  mux.HandleFunc("GET /ping", handler)

  req := httptest.NewRequest(http.MethodGet, "/ping", nil)
  rec := httptest.NewRecorder()

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    rec.Body.Reset()
    mux.ServeHTTP(rec, req)
  }
}
```

The standard library establishes the baseline execution model. A request is parsed, routed through `ServeMux`, passed directly to a handler function, and written to a response writer. There is very little indirection and almost no hidden state. Each request allocates what is required for request parsing and response writing, and nothing more unless user code introduces it.

There is no context pooling, no handler wrapping beyond what is strictly necessary, and no framework level bookkeeping. This makes `net/http` predictable and easy to reason about. Its performance characteristics are stable and transparent, which is why it is often used as a reference point when evaluating other frameworks.

Chi

```
package bench

import (
  "net/http"
  "net/http/httptest"
  "testing"

  "github.com/go-chi/chi/v5"
)

func BenchmarkChi(b *testing.B) {
  r := chi.NewRouter()
  r.Get("/ping", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("pong"))
  })

  req := httptest.NewRequest(http.MethodGet, "/ping", nil)
  rec := httptest.NewRecorder()

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    rec.Body.Reset()
    r.ServeHTTP(rec, req)
  }
}
```

Chi adds a routing and middleware layer on top of `net/http` while preserving the same core request and response types. Each request traverses a route tree, resolves parameters, and walks a middleware chain before reaching the handler. This adds a small amount of overhead compared to raw `ServeMux`.

Because Chi does not pool request contexts or response writers, allocations are slightly higher than the standard library but still modest. The cost comes primarily from route matching and middleware traversal rather than from object management. Chi's performance model favors clarity and composability over aggressive optimization, which keeps behavior predictable even as complexity grows.

Gin

```
package bench

import (
  "net/http"
  "net/http/httptest"
  "testing"
```

```
  "github.com/gin-gonic/gin"
)

func BenchmarkGin(b *testing.B) {
  gin.SetMode(gin.ReleaseMode)

  r := gin.New()
  r.GET("/ping", func(c *gin.Context) {
    c.String(200, "pong")
  })

  req := httptest.NewRequest(http.MethodGet, "/ping", nil)
  rec := httptest.NewRecorder()

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    rec.Body.Reset()
    r.ServeHTTP(rec, req)
  }
}
```

Gin introduces a custom context type that is pooled and reused across requests. On each request, a context object is pulled from a pool, its internal state is reset, middleware is executed, and response metadata such as status and size is tracked explicitly.

This pooling strategy reduces allocations compared to frameworks that create fresh context objects for every request. The tradeoff is additional bookkeeping to reset state correctly and manage lifecycle transitions. For trivial handlers with no middleware, this overhead can make Gin slightly slower than raw `net/http`. As middleware stacks grow, the pooled model often becomes more efficient because it amortizes allocation costs.

Gin's performance profile is shaped more by its lifecycle management than by routing complexity.

Echo

```
package bench

import (
  "net/http"
  "net/http/httptest"
  "testing"

  "github.com/labstack/echo/v4"
)

func BenchmarkEcho(b *testing.B) {
  e := echo.New()
  e.GET("/ping", func(c echo.Context) error {
    return c.String(200, "pong")
  })
```

```
  req := httptest.NewRequest(http.MethodGet, "/ping", nil)
  rec := httptest.NewRecorder()

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    rec.Body.Reset()
    e.ServeHTTP(rec, req)
  }
}
```

Echo uses a model similar to Gin, with a custom context abstraction and internal state tracking. Context objects are reused, handlers are invoked through interface calls, and response state is captured by the framework rather than inferred from the response writer.

The cost per request includes context reuse, handler dispatch, and error handling hooks. In practice, Echo's performance is usually close to Gin's. Differences tend to appear when error handling, logging, or recovery behavior is enabled, rather than in the routing or handler invocation itself.

Fiber

```
package bench

import (
  "io"
  "net/http"
  "testing"

  "github.com/gofiber/fiber/v2"
)

func BenchmarkFiber(b *testing.B) {
  app := fiber.New()
  app.Get("/ping", func(c *fiber.Ctx) error {
    return c.SendString("pong")
  })

  req, _ := http.NewRequest(http.MethodGet, "/ping", nil)

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    resp, _ := app.Test(req)
    io.ReadAll(resp.Body)
    resp.Body.Close()
  }
}
```

Fiber is built on `fasthttp`, which uses a fundamentally different execution model than `net/http`. It avoids `context.Context`, aggressively reuses objects, and buffers responses by default. This minimizes allocations and system calls for simple request paths.

As a result, Fiber often shows very high throughput and low allocation counts in microbenchmarks. However, these numbers reflect a different set of tradeoffs. Compatibility with standard libraries is reduced, streaming semantics differ, and many tools designed for `net/http` cannot be used directly.

Benchmarks that compare Fiber directly to `net/http` are not measuring the same thing. They compare two different network stacks with different guarantees and expectations.

Mizu

```
package bench

import (
  "net/http"
  "net/http/httptest"
  "testing"

  "github.com/go-mizu/mizu"
)

func BenchmarkMizu(b *testing.B) {
  app := mizu.New()
  app.Get("/ping", func(c *mizu.Ctx) error {
    return c.Text(200, "pong")
  })

  req := httptest.NewRequest(http.MethodGet, "/ping", nil)
  rec := httptest.NewRecorder()

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    rec.Body.Reset()
    app.ServeHTTP(rec, req)
  }
}
```

Mizu stays aligned with `net/http` while introducing a thin context layer and structured middleware pipeline. Each request creates a context, traverses middleware, executes the handler, and runs error handling hooks. There is no aggressive pooling by default, and the focus is on correctness, transparency, and predictable behavior.

The additional cost compared to raw `net/http` comes from explicit lifecycle management rather than hidden buffering or object reuse. In practice, performance is usually close to Chi and standard library based routers, especially once middleware is present.

How to read benchmark numbers

Benchmarks only measure what they execute. A minimal handler benchmark measures routing and dispatch overhead, not database access, serialization, or network latency. Numbers are only comparable when frameworks are configured with similar middleware, similar response behavior, and similar network stacks.

Comparing `fasthttp` based frameworks to `net/http` based ones mixes different abstractions. Looking only at nanoseconds per operation hides allocation behavior, which often matters more under load. The most reliable benchmarks are those that include real handlers and realistic workloads.

In most production systems, framework overhead is dwarfed by I/O, database access, and external services. The difference between frameworks becomes relevant only when the rest of the system is already well optimized.

What learners should focus on

Performance reflects priorities. Some frameworks optimize for raw throughput by reusing aggressively and limiting abstractions. Others prioritize compatibility, clarity, and predictable behavior. Buffering simplifies APIs but affects streaming. Pooling reduces allocations but increases lifecycle complexity.

The most useful framework is not the one with the best microbenchmark result, but the one whose performance model you understand well enough to reason about behavior under real load.

net/http interoperability

Interoperability answers a practical, long term question: can a framework participate naturally in the Go HTTP ecosystem, or does it require adapters, shims, or architectural boundaries to coexist with existing code.

In Go, interoperability is not an abstract concept. It is largely determined by whether a framework aligns with `net/http` contracts. If a framework is an `http.Handler`, it can be mounted into a standard `http.Server`. If it accepts `http.Handler`, it can host existing libraries. If its middleware model matches `net/http`, middleware can be reused without translation. When these properties hold, third party libraries work unchanged and systems remain composable.

This section evaluates interoperability through concrete checks. A standard `http.Handler` is mounted inside the framework. The framework itself is mounted inside a standard `http.Server`. Existing `net/http` middleware is reused without rewriting. These checks reveal how much friction exists at the integration boundary.

net/http

```
package main

import (
  "net/http"
)

func stdHandler(w http.ResponseWriter, r *http.Request) {
  w.Write([]byte("from std handler\n"))
}

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/std", stdHandler)
```

```
  http.ListenAndServe(":8080", mux)
}
```

The standard library defines the reference model. Everything revolves around `http.Handler`. Servers accept handlers, routers are handlers, and middleware is just a function that wraps a handler and returns another handler.

Because this is the base abstraction, all `net/http` libraries are interoperable by definition. There is no adaptation cost, no translation layer, and no impedance mismatch. This simplicity is why `net/http` remains the foundation of most Go web systems, even when higher level frameworks are used on top.

Chi

```
package main

import (
  "net/http"

  "github.com/go-chi/chi/v5"
)

func stdHandler(w http.ResponseWriter, r *http.Request) {
  w.Write([]byte("from std handler\n"))
}

func main() {
  r := chi.NewRouter()

  r.Handle("/std", http.HandlerFunc(stdHandler))

  http.ListenAndServe(":8080", r)
}
```

Chi preserves the `net/http` contract completely. The router itself implements `http.Handler`, and routes accept standard handlers directly. This means a Chi router can be mounted anywhere a standard handler is expected, and any standard handler can be mounted inside Chi without modification.

Middleware compatibility follows naturally. Middleware written for `net/http` can wrap a Chi router, and Chi middleware can wrap standard handlers. Third party libraries that expect `http.Handler` work unchanged. From an interoperability perspective, Chi behaves as a structured extension of `net/http`, not a replacement.

This property makes Chi particularly easy to introduce into existing codebases incrementally.

Gin

```
package main

import (
```

```
  "net/http"

  "github.com/gin-gonic/gin"
)

func stdHandler(w http.ResponseWriter, r *http.Request) {
  w.Write([]byte("from std handler\n"))
}

func main() {
  r := gin.New()

  r.GET("/std", gin.WrapH(http.HandlerFunc(stdHandler)))

  http.ListenAndServe(":8080", r)
}
```

Gin implements `http.Handler`, so it can be mounted inside a standard `http.Server`. This allows Gin applications to run within the `net/http` server infrastructure without issue.

However, Gin does not accept standard handlers directly. Handler signatures use `*gin.Context`, so standard handlers must be adapted using `gin.WrapH`. This adapter bridges between Gin's context model and `http.ResponseWriter` and `*http.Request`.

The result is functional but not seamless. Standard middleware written for `net/http` cannot be reused directly inside Gin without similar adapters. Interoperability exists, but it lives behind explicit wrapping boundaries. Gin remains close to `net/http`, but it does not sit fully on the same abstraction path.

Echo

```
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
)

func stdHandler(w http.ResponseWriter, r *http.Request) {
  w.Write([]byte("from std handler\n"))
}

func main() {
  e := echo.New()

  e.GET("/std", echo.WrapHandler(http.HandlerFunc(stdHandler)))

  http.ListenAndServe(":8080", e)
}
```

Echo also implements `http.Handler`, which allows it to be served by a standard `http.Server`. Like Gin, Echo uses its own handler signature and context abstraction, so standard handlers must be adapted before being mounted.

Echo provides explicit adapter functions such as `WrapHandler`, which makes the interoperability boundary clear and intentional. Net/http middleware cannot be reused directly without adaptation, but the required translation is well defined and supported.

Echo's interoperability model is explicit rather than implicit. Integration is possible, but the framework does not attempt to hide the boundary between its abstractions and the standard library.

Fiber

```
package main

import (
  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()

  app.Get("/std", func(c *fiber.Ctx) error {
    return c.SendString("from fiber\n")
  })

  app.Listen(":8080")
}
```

Fiber is not built on `net/http`. It does not implement `http.Handler`, and it cannot be mounted inside a standard `http.Server`. Its handler and middleware model is entirely separate, built on top of `fasthttp`.

As a result, standard `net/http` handlers and middleware cannot be reused directly. Integration with `net/http` based systems requires adapters, reverse proxies, or running separate servers. This is a deliberate design choice that prioritizes performance and control over compatibility.

Fiber represents a clean break from the Go HTTP ecosystem rather than an extension of it.

Mizu

```
package main

import (
  "net/http"

  "github.com/go-mizu/mizu"
)

func stdHandler(w http.ResponseWriter, r *http.Request) {
  w.Write([]byte("from std handler\n"))
}
```

```
func main() {
  app := mizu.New()

  app.Handle("/std", http.HandlerFunc(stdHandler))

  http.ListenAndServe(":8080", app)
}
```

Mizu is designed to remain fully on the `net/http` path. It implements `http.Handler`, accepts standard handlers directly, and can be wrapped by standard middleware without translation. Existing `net/http` libraries work unchanged when embedded in a Mizu application.

At the same time, Mizu provides higher level abstractions such as structured routing, middleware chaining, and context helpers without replacing the underlying contracts. This allows applications to grow in complexity while preserving interoperability with the broader ecosystem.

Interoperability is treated as a first class constraint rather than a compatibility layer.

What learners should focus on

Interoperability determines how easily a system evolves over time. Frameworks that preserve `net/http` contracts allow code to be reused across services, libraries to be shared without adapters, and components to be composed freely. They also make it possible to remove or replace the framework later without rewriting the entire application.

Frameworks that diverge from `net/http` can offer performance or ergonomics benefits, but they narrow the integration surface. Understanding where a framework sits on this spectrum is essential for making architectural decisions that remain flexible as systems grow.

Tradeoffs

Tradeoffs are easiest to understand when you see **what code you write** and **what code you cannot avoid writing**. This chapter therefore starts with minimal, comparable code snippets and then synthesizes the differences in a single table, followed by a deep dive that explains why those differences matter over time.

The goal is not to repeat comparisons, but to surface **structural tradeoffs** that persist even as frameworks add features or polish APIs.

net/http

```
package main

import (
  "net/http"
)

func ping(w http.ResponseWriter, r *http.Request) {
  w.Write([]byte("pong"))
```

```
}

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("GET /ping", ping)

  http.ListenAndServe(":8080", mux)
}
```

This is the reference model. Handlers are plain functions. Routing is explicit. Middleware is simple wrapping. There is no hidden lifecycle and no framework state. Everything composes through `http.Handler`. The tradeoff is that nothing is provided for you beyond primitives. Structure, consistency, and safety are your responsibility.

Chi

```
package main

import (
  "net/http"

  "github.com/go-chi/chi/v5"
)

func main() {
  r := chi.NewRouter()
  r.Get("/ping", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("pong"))
  })

  http.ListenAndServe(":8080", r)
}
```

Chi preserves the net/http contract while adding structure. Handlers remain plain functions, middleware is still wrapping, and routing becomes expressive. The tradeoff is a small amount of routing and middleware overhead in exchange for clarity and composability. Chi does not try to manage lifecycle or state for you, so behavior remains visible and testable.

Gin

```
package main

import (
  "net/http"

  "github.com/gin-gonic/gin"
)

func main() {
  r := gin.New()
```

```
  r.GET("/ping", func(c *gin.Context) {
    c.String(http.StatusOK, "pong")
  })

  http.ListenAndServe(":8080", r)
}
```

Gin replaces the handler contract. Handlers are no longer plain functions but methods operating on a pooled context object. This enables convenience APIs and reduces allocations, but it also couples application code to Gin's lifecycle. The tradeoff is ergonomics and performance versus isolation and portability. Testing and middleware reuse must flow through Gin's engine.

Echo

```
package main

import (
  "net/http"

  "github.com/labstack/echo/v4"
)

func main() {
  e := echo.New()
  e.GET("/ping", func(c echo.Context) error {
    return c.String(http.StatusOK, "pong")
  })

  http.ListenAndServe(":8080", e)
}
```

Echo makes a similar tradeoff to Gin, but keeps a slightly looser abstraction. Handlers still depend on a framework context, but can be tested in isolation by constructing that context manually. The tradeoff sits between Gin and Chi: more structure and batteries included than Chi, but less hidden lifecycle than Gin.

Fiber

```
package main

import (
  "github.com/gofiber/fiber/v2"
)

func main() {
  app := fiber.New()
  app.Get("/ping", func(c *fiber.Ctx) error {
    return c.SendString("pong")
  })

  app.Listen(":8080")
```

```
}
```

Fiber opts out of the net/http contract entirely. The execution model, context, middleware, and networking stack are all framework specific. This enables aggressive reuse and high throughput, but it also isolates the application from the standard Go HTTP ecosystem. The tradeoff is performance and simplicity versus interoperability and reuse.

Mizu

```go
package main

import (
  "net/http"

  "github.com/go-mizu/mizu"
)

func main() {
  app := mizu.New()
  app.Get("/ping", func(c *mizu.Ctx) error {
    return c.Text(http.StatusOK, "pong")
  })

  http.ListenAndServe(":8080", app)
}
```

Mizu deliberately stays on the net/http path while adding a structured request model. Handlers use a context abstraction, but the framework itself remains an `http.Handler` and preserves standard request semantics. The tradeoff is accepting a thin abstraction layer in exchange for consistency across routing, middleware, observability, and testing without breaking ecosystem compatibility.

Tradeoffs at a glance

Axis	net/http	Chi	Gin	Echo	Fiber	Mizu
Handler contract	Standard	Standard	Custom	Custom	Custom	net/http-aligned
Context type	`context.Context`	`context.Context`	Pooled custom	Custom	Custom	net/http + thin ctx
Middleware reuse	Native	Native	Adapter	Adapter	None	Native
Testing granularity	Max	Max	Router-only	Mixed	Router-only	Max + pipeline
Lifecycle visibility	Explicit	Explicit	Managed	Semi-managed	Managed	Explicit
Ecosystem compatibility	Full	Full	Partial	Partial	Low	Full
Performance bias	Predictable	Predictable	Pooled	Pooled	Aggressive	Predictable

Deep dive: why these tradeoffs matter

The most durable tradeoff is **contract preservation versus replacement**. Once a framework replaces the handler contract, all application code becomes framework specific. This affects not only routing, but testing, middleware sharing, observability integration, and long term refactoring. Preserving the contract keeps options open.

The second critical tradeoff is **lifecycle visibility**. When request boundaries are explicit, instrumentation, error handling, and debugging are straightforward. When lifecycle is managed internally, behavior becomes easier to use but harder to reason about when something goes wrong. This difference becomes significant in production systems where observability and correctness matter more than convenience.

Performance tradeoffs are often misunderstood. Pooling and buffering can improve microbenchmarks, but they also introduce constraints on how code can be written and tested. In most real systems, framework overhead is dominated by I/O and business logic. Predictability is usually more valuable than peak throughput.

Finally, interoperability determines how systems evolve. Frameworks that remain compatible with net/http can be introduced gradually, coexist with other components, and be removed later if needed. Frameworks that replace core contracts should be chosen deliberately, with a clear understanding that the boundary they introduce is permanent.

The correct framework is therefore not the one with the most features, but the one whose tradeoffs align with the lifetime and integration needs of the system you are building.